PhD-FSTM-2023-096
The Faculty of Science, Technology and Medicine

Dissertation

Defence held on 15/09/2023 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

Haoye Tian
Born on 11^{th} September 1993 in Henan (China)

Learning Code Change Semantics for Patch Correctness Assessment in Program Repair

Dissertation Defence Committee

Dr. Tegawendé F. Bissyandé, Dissertation Supervisor
Associate Professor, University of Luxembourg

Dr. Jacques Klein, Chairman
Full Professor, University of Luxembourg

Dr. Maxime Cordy, Vice Chairman
Research Scientist, University of Luxembourg

Dr. Claire Le Goues
Associate Professor, Carnegie Mellon University

Dr. David Lo
Full Professor, Singapore Management University

Abstract

The growing complexity of modern software systems and the demand for accelerated development cycles have made software defects more prevalent and challenging to repair, necessitating effective and efficient solutions. Automated Program Repair (APR) has been proposed as a potential approach to address these issues by automatically identifying and repairing software defects, leveraging techniques from artificial intelligence, data mining, symbolic execution, and formal methods. The practical adoption of APR techniques however faces a significant challenge due to patch overfitting. In the absence of perfect program specifications and test oracle, APR tools frequently depend on incomplete test suites or weak constraints as approximations for assessing patch correctness, which may lead to the generation of patches that simply overfit to the weak oracle but do not generalize in practical production. Indeed, these overfitting patches do not implement the intended behavior that developers expect, highlighting the need for enhanced solutions in the field of APR.

State-of-the-art APR techniques currently produce patches that are manually evaluated as overfitting, and these overfitting patches often worsen the original program, leading to negative effects such as introducing security vulnerabilities and removing useful features. This obstructs the development of APR techniques that rely on feedback from correctly generated patches, and the expense of developers' manual debugging has shifted to evaluating patch correctness. Automated assessment of patch correctness has the potential to reduce patch validation costs and accelerate the identification of practically correct patches, making it easier for developers to adopt APR techniques. While the proposed approaches have been demonstrated to be effective in the literature, several challenges remain unexplored and warrant further investigation.

This thesis begins with an empirical analysis of a prevalent hypothesis concerning patch correctness, leading to the establishment of a patch correctness prediction framework based on representation learning. Second, we propose to validate correct patches by proposing a novel heuristic on the relationship between patches and their associated failing test cases. Lastly, we present a novel perspective to assess patch correctness with natural language processing. Our contributions to the research field through this thesis are as follows: 1) assessing the feasibility of utilizing advancements in deep representation learning to generate patch embeddings suitable for reasoning about correctness. Consequently, we establish Leopard, a supervised learning-based patch correctness prediction framework. 2) comparing code embeddings and engineered features for patch correctness prediction, and investigating their combination in Panther (an upgraded version of Leopard) for more accurate classification. Additionally, we use the SHAP explainability model to reveal the essential aspects of patch correctness by interpreting underlying causes of prediction performance across features and classifiers. 3) presenting and validating a key

hypothesis: when different programs fail to pass similar test cases, it is likely that these programs require similar code changes. Based on this heuristic, we propose BATS, an approach predicting patch correctness by statically comparing generated patches against previous correct patches failing on similar tests. 4) proposing a novel perspective to patch correctness assessment: a correct patch implements changes that *answer* to the issue caused by the buggy behavior. By leveraging bug reports to offer an explicit description of the bug, we build QUATRAIN, a supervised learning approach that utilizes a deep NLP model to predict the relevance between a bug report and a patch description.

Plato is dear to me, but dearer still is truth.

Aristotle

Acknowledgements

I would like to convey my deepest gratitude to those who have generously shared their invaluable knowledge, guidance, and experiences during my PhD journey. Their support was pivotal, without which the completion of my dissertation would not have been feasible. I am truly honored to have embarked on this scholarly journey alongside them.

First, I am incredibly grateful to Prof. Tegawendé F. Bissyandé, my supervisor, who has given me the remarkable opportunity to pursue my doctoral degree across continents. Throughout my PhD journey, he consistently provided invaluable support and guidance, not only in my research but also in matters pertaining to life and career. As an educator, his selflessness and discipline continue to inspire me, leaving an indelible mark on my personal growth.

Second, I am equally grateful to my daily adviser, Prof. Jacques Klein, who is open to giving valuable advice and discussion for my research. I am particularly grateful to Dr. Kui Liu, whose regular mentorship has empowered me to conduct research and write technical papers. I extend profound thanks to Prof. Shing-Chi Cheung, who graciously welcomed me to the Hong Kong University of Science and Technology for an enriching academic visit, offering precious advice and another picture of research. Their dedicated guidance has transformed my PhD journey into a fruitful and fulfilling experience, and I am genuinely grateful for the friendships we have cultivated over the years.

Third, I would like to extend my thanks to all my co-authors including Prof. Li Li, Prof. Xin Xia, Dr. Andrew Habib, Pingfan Kong, Weiguo Pian, Xunzhu Tang, Abdoul Kader Kaboré, etc., for their helpful discussions and collaborations.

I want to thank the members of my Ph.D. defence committee, including chairman Prof. Jacques Klein, vice-chairman Dr. Maxime Cordy, Prof. Claire Le Goues, Prof. David Lo, and my supervisor Prof. Tegawendé F. Bissyandé. It is my great honor to have them on my defence committee, and I appreciate very much their efforts to review my dissertation and evaluate my Ph.D. work.

With great pleasure, I express my thanks to all members of TruX and friends that I have made in the Grand Duchy of Luxembourg for the memorable moments. In particular, I would like to thank my girlfriend, Ziyun Zhou, for her love and support.

Finally, I can never overemphasize the importance of the support and love from my parents. Their belief in the power of education has been instrumental in shaping my character and accomplishments today.

Haoye Tian
University of Luxembourg
August 2023

Contents

List of Figures

List of Tables

1 Introduction

In this chapter, we first introduce the problem of patch overfitting in automated program repair. Then, we describe the limitations and challenges researchers face when addressing patch overfitting. Finally, we present the contributions and the roadmap of this dissertation.

Contents

1.1 Motivation

The rapid expansion of software systems across various industries and aspects of everyday life has underscored the importance of ensuring the reliability and maintainability of these systems. However, the complexity of modern software, combined with the mounting pressure for accelerated development cycles, has rendered software defects not only inevitable but also more numerous and harder to repair, presenting a serious threat to the prospects of software companies and developers [5, 6]. A notable example is that a software bug in Knight Capital Group's trading algorithm caused unintended trades, leading to a $440 million loss in 45 minutes, which ultimately resulted in the company's acquisition in 2012 [7]. Meanwhile, manual debugging becomes increasingly expensive as it grows more labor-intensive and time-consuming [8]. Consequently, the demand for effective and efficient solutions to address these challenges has grown substantially. Automated Program Repair (APR) [9, 10, 11] has emerged as a potential approach to tackle this problem by automatically identifying and repairing software defects, reducing the burden on developers and elaborating the debugging process, ultimately contributing to the development of more reliable, trustworthy, and robust software systems.

APR has achieved new milestones by generating valid patches for various defects, leveraging cutting-edge techniques from artificial intelligence [12, 13, 14, 15], data mining [16, 17, 18, 19], symbolic execution [20, 21, 22], and formal methods [23, 24]. While the techniques are effective, their adoption by industry faces a crucial challenge with respect to their practicality: patch overfitting problem [25, 26]. In practice, due to the absence of the ideal program oracle or specifications, existing APR tools typically rely on crafted test suites or constraints as a cost-effective approximation for evaluating patch correctness. As a result, they tend to produce patches that overfit to weak oracle (e.g., test suites), failing to generalize to independent real-world applications. In other words, the patched program, although passing test cases, are still incorrect. In essence, these generated patches do not implement the intended behavior that developers expect from the program. Such an inaccurate patch is usually referred to as an *overfitting patch*, while a patch that passes all test cases in the test suite is known as a *plausible patch* [27, 25].

Currently, most, if not all, plausible patches produced by state-of-the-art APR techniques are verified manually as overfitting [28, 29, 30]. More importantly, Smith et al. [26] discovered that these overfitting patches frequently worsen the patched program compared to the unpatched versions due to their negative effects, such as introducing security vulnerabilities and removing useful features. The overwhelming majority of overfitting patches obstruct the development of APR techniques that rely on feedback from correctly generated patches. From the perspective of developers, the expense of manual debugging has not disappeared; instead, it has shifted towards evaluating patch correctness. Furthermore, a recent survey [31] revealed that a mere 22% of them are willing to review up to 10 automatically generated patches before abandoning the adoption of the technique.

Automated assessment of patch correctness [32, 33, 27] holds the promise of reducing patch validation costs by automatically identifying correct patches among all plausible patches. In production, patch correctness assessment enhances the effectiveness and efficiency of APR techniques by boosting the identification and generation of practically correct patches. In application, prioritizing correct patches from

numerous plausible yet incorrect patches eases developers' efforts and strengthens their trust in adopting APR techniques. While patch correctness assessment methods have shown effectiveness, there are still challenges that need further investigation.

In this dissertation, we propose to devise novel patch correctness assessment techniques for two purposes: 1) to explore static approaches that efficiently address patch overfitting; and 2) to identify the behavior of a correct patch that is relevant to the bug targeted.

1.2 Limitations and Challenges

In this section, we describe the limitations of existing approaches that tackle patch overfitting, with an emphasis on their efficiency. Subsequently, we present the crucial challenge in assessing patch correctness: identifying the behavior of correct patches.

1.2.1 Limitations of Existing Approaches

Numerous existing approaches in the literature [27, 34, 35, 2, 1, 36] have advocated for dynamic approaches to mitigate patch overfitting by utilizing dynamic information generated during program execution. Among them, one hypothesis is that augmenting weak oracle can help reduce the generation of overfitting patches. Following up on the intuition, researchers have suggested enhancing the quality and quantity of test cases to filter out incorrect patches in advance [34, 35, 1, 37]. However, they have the test oracle problem, that is we do not always have an accurate specification of what the output should be. Alternatively, heuristic-based methods have been explored to differentiate between the execution paths of test cases influenced by correct and overfitting patches (runtime analysis) [2]. Despite the promising results demonstrated, dynamic approaches require more computational resources and time-consuming execution, as they need to adapt to new data and changes at runtime. This, therefore, hinders their application for efficiently validating a large number of plausible patches. To sum it up, dynamic approaches are practically infeasible.

Static approaches have garnered significant interest from researchers, as they do not typically depend on dynamic execution, resulting in a faster patch validation process [38, 27, 39, 40]. Utilizing manually crafted anti-patterns, researchers have suggested filtering out overfitting patches that implement prohibited modifications [40]. Additionally, advancements in artificial intelligence have led researchers to explore the identification and classification of correct patches using machine/deep learning models trained on statically extracted features (e.g., ASE differencing or P4J) of code or patches [41, 27]. Although these methods circumvent the time expense associated with program execution, they still entail manual effort in extracting knowledge (patterns and features) relevant to patch correctness. Furthermore, this manual knowledge may not be scalable across various scenarios, especially if it is derived solely from specific domains or datasets. In summary, existing approaches for addressing patch overfitting tend to be high-cost and this violates the original purpose of APR: automation and efficiency. To mitigate the limitations, in this dissertation, we propose heuristic-based static (un)supervised approaches that automatically learn to identify patch correctness by capitalizing on advances in machine/deep learning learners and pre-trained large-scale language models (LLM).

1.2.2 Challenges in Capturing Correct Patch Behavior

How to capture the semantics (a.k.a intention) of a patch remains a significant challenge in patch correctness assessment. In the existing literature, hypotheses regarding patch correctness typically focus on the change to the original code caused by the patch. A widespread assumption is that bug fixes generally result in minimal changes [42, 43, 44, 45, 46, 18]. Based on this assumption, researchers have explored measurements of syntactic change in patches, such as AST node distance [41, 46] and invariants difference [47, 48], to prioritize smaller patches, which are more likely to be correct. Furthermore, researchers attempted to capture the behavior of code changes

by either extracting historical patterns [40, 49] that (in)correct patches (violate) adhere to or by observing differences in dynamic test executions [2] between the original and patched programs. Nevertheless, the extracted code change semantics remain limited in practice, and the deep representation of the patch itself has not been fully exploited. For instance, devising anti-patterns [40, 49] to represent all the behavior of the overfitting patches is impossible for developers in real-world scenarios. More importantly, few of the previous studies directly or explicitly investigate the correct behavior of the patches that is relevant to the bug it aims to address. As previously stated, the cause of patch overfitting is the failure of patches to implement the intended correct behavior. Therefore, capturing correct patch behavior in relation to the bug is necessarily crucial for patch correctness assessment in program repair. As a matter of fact, correct patch behavior is practically implicit and difficult to discern.

In general, while the challenges present difficulties, they also open doors to opportunities. Seizing the opportunities, we proposed four research focuses, which are static supervised or unsupervised learning approaches to assess patch correctness. We employ LLMs in this dissertation to capture the behavior of the patches (code change semantics) and assess their correctness by heuristically correlating the behavior with the bug (which can be represented as failing test cases and bug reports).

1.3 Contributions

In this chapter, we provide a summary of the contributions of this dissertation as follows:

- **LEOPARD: Evaluating representation learning of code changes for predicting patch correctness in program repair.** Embeddings, through representation learning, have been effectively employed in numerous prediction tasks within software engineering research. However, the existing literature lacks comprehensive experimental results for patch correctness prediction, leaving room for further exploration. We study the benefit of learning code representations in order to learn deep features that may encode the properties of patch correctness. Our empirical work mainly investigates different representation learning approaches for code changes to derive embeddings that are amenable to similarity computations. We report on findings based on embeddings produced by pre-trained and re-trained neural networks. Experimental results demonstrate the potential of embeddings to empower learning algorithms in reasoning about patch correctness: a machine learning predictor with BERT transformer-based embeddings associated with logistic regression yielded an AUC value of about 0.8 in the prediction of patch correctness on a deduplicated dataset of 1000 labeled patches. Our investigations show that learned representations can lead to reasonable performance when comparing against the state-of-the-art approach which relies on dynamic information.

  This work has led to a research paper published to the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020).

- **PANTHER: Combining learned embeddings with engineered features for accurate prediction of correct patches.** The literature demonstrates the potential of learned embeddings for patch correctness reasoning. However, it is unclear which feature type—learned embeddings or engineered features—performs better in predicting correct patches and whether their combination can enhance performance. By combining deep learned embeddings and engineered features, we propose Panther (the upgraded version of Leopard implemented in this work), which outperforms Leopard with higher scores in terms of AUC, +Recall and -Recall, and can accurately identify more (in)correct patches that cannot be predicted by the classifiers only with learned embeddings or engineered features. Additionally, we use an explainable ML technique, SHAP, to empirically interpret how the learned embeddings and engineered features are contributed to the patch correctness prediction.

  This work has led to a research paper published to the ACM Transactions on Software Engineering and Methodology in 2023 (TOSEM 2023).

- **BATS: An unsupervised learning-based approach to predict patch correctness by checking patch Behaviour Against failing Test Specification.** APR systems struggle to address the problem of patch overfitting, given the incompleteness of available test suites. Our intuition is that we can triage correct patches by checking whether each generated patch implements code changes (i.e., behaviour) that are relevant to the bug it addresses. Such a bug is commonly specified by a failing test case. Towards predicting patch correctness in APR, we propose a novel yet simple hypothesis on how the

link between the patch behaviour and failing test specifications can be drawn: similar failing test cases should require similar patches. We then propose BATS, an unsupervised learning-based approach to predict patch correctness by checking patch Behaviour Against failing Test Specification. BATS exploits deep representation learning models for code and patches: for a given failing test case, the yielded embedding is used to compute similarity metrics in the search for historical similar test cases to identify the associated applied patches, which are then used as a proxy for assessing the correctness of the APR-generated patches. Experimentally, we first validate our hypothesis by assessing whether ground-truth developer patches cluster together in the same way that their associated failing test cases are clustered. Then, after collecting a large dataset of 1,278 plausible patches (written by developers or generated by 32 APR tools), we use BATS to predict correct patches: BATS achieves AUC between 0.557 to 0.718 and recall between 0.562 and 0.854 in identifying correct patches. Our approach outperforms state-of-the-art techniques for identifying correct patches without the need for large labeled patch datasets; as is the case with machine learning-based approaches. While BATS is constrained by the availability of similar test cases, we show that it can still be complementary to existing approaches: when combined with a recent approach that relies on supervised learning, BATS improves the overall recall in detecting correct patches. We finally show that BATS is complementary to the state-of-the-art PATCH-SIM dynamic approach for identifying correct patches generated by APR tools.

This work has led to a research paper published to the ACM Transactions on Software Engineering and Methodology in 2022 (TOSEM 2022).

- **QUATRAIN: Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness.** State-of-the-art approaches generally validate patch correctness by reasoning the code changes and the test suites. However, the bug targeted by the generated patch is rarely explicitly explored, despite patches being designed to address specific buggy behaviors. We propose a novel perspective to the problem of patch correctness assessment: a correct patch implements changes that "answer" to a problem posed by buggy behavior. Concretely, we turn the patch correctness assessment into a Question Answering problem. To tackle this problem, our intuition is that natural language processing can provide the necessary representations and models for assessing the semantic correlation between a bug (question) and a patch (answer). Specifically, we consider as inputs the bug reports as well as the natural language description of the generated patches. Our approach, QUATRAIN, first considers state-of-the-art commit message generation models to produce the relevant inputs associated to each generated patch. Then we leverage a neural network architecture to learn the semantic correlation between bug reports and commit messages. Experiments on a large dataset of 9 135 patches generated for three bug datasets (Defects4j, Bugs.jar and Bears) show that QUATRAIN achieves an AUC of 0.886 on predicting patch correctness, and recalling 93% correct patches while filtering out 62% incorrect patches. Our experimental results further demonstrate the influence of inputs quality on prediction performance. We further perform experiments to highlight that the model indeed learns the relationship between bug reports and code change descriptions for the

prediction. Finally, we compare against prior work and discuss the benefits of our approach.

This work has led to a research paper published to the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022).

1.4 Roadmap

The roadmap of this dissertation is illustrated in Figure 1.1. Chapter 2 introduces the background of automated program repair, patch overfitting, and related work, followed by a summary of literature addressing patch overfitting. Chapter 3 presents an empirical study evaluating representation learning of code changes and proposes a patch correctness prediction framework. Building upon this study, Chapter 4 develops an upgraded version of the framework (PANTHER) by combining learned and engineered features of the patch, as well as offering an analysis of the explanations behind the predictions. Chapter 5 introduces a novel hypothesis and approach, BATS, which assesses patch correctness by identifying correct patch behavior based on similarities between failing test cases and patches. In Chapter 6, we propose a novel perspective on the problem of patch correctness assessment and present QUATRAIN, an approach that evaluates patch correctness by learning the semantic correlation between the bug report and the commit message of the patch. Chapter 7 concludes the dissertation, and Chapter 8 discusses future work.

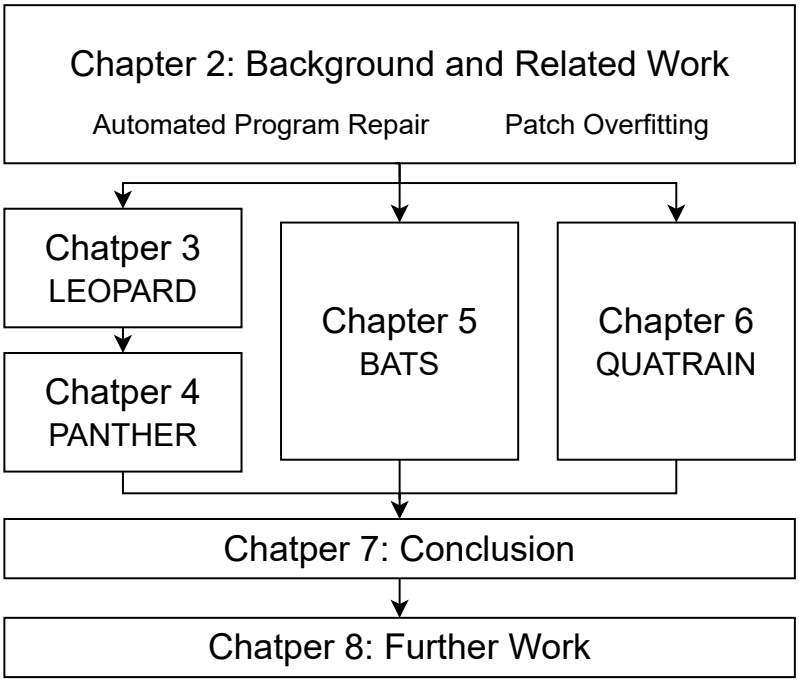


Figure 1.1: Roadmap of this dissertation.

2 Background and Related Work

This chapter introduces the necessary concepts and related works for understanding the objective, techniques, contributions and key concerns of the research studies that we have conducted in this dissertation. Specifically, we present some technical details and analysis of automated program repair and patch overfitting, respectively.

Contents

2.1 Automated Program Repair

Automated Program Repair (APR) is an emerging domain in software engineering, focusing on the automated detection and rectification of bugs or errors in software programs. Utilizing advanced approaches, including heuristics, machine learning, and LLMs, APR aims to enhance software development, maintenance, and quality assurance processes while minimizing the resources required for manual debugging. The generate-and-validate APR methodology generally comprises three stages: fault localization, patch generation, and patch validation, as illustrated in Figure 2.1 and detailed in the subsequent definitions. We present an overview of representative APR tools, organized according to three prevalent categories: heuristic-based repair, constraint-based repair, and learning-based repair [10].

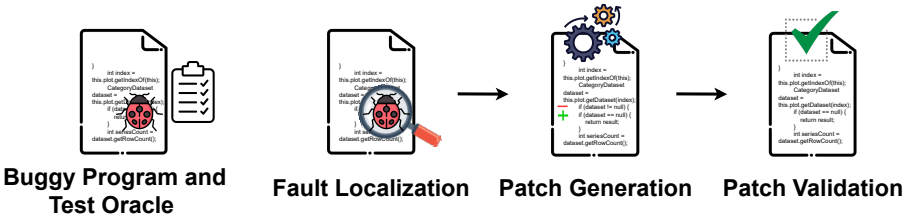


Figure 2.1: The pipeline of generate-and-validate program repair.

Definition 2.1.1 (Buggy Program) *A buggy program is represented by the pair* $(\mathcal{P}, \mathcal{O})$*, where* $\mathcal{P}$ *represents a program that fails to satisfy explicit or implicit oracle* $\mathcal{O}$*.*

Definition 2.1.2 (Fault Localization) *Given a buggy program* $(\mathcal{P}, \mathcal{O})$*, fault localization is the systematic process of identifying the specific fault location(s)* $\mathcal{L}$ *within the* $(\mathcal{P}, \mathcal{O})$*.*

Definition 2.1.3 (Patch Generation) *Given a buggy program* $(\mathcal{P}, \mathcal{O})$ *with identified fault location(s)* $\mathcal{L}$ *that fails to meet oracle* $\mathcal{O}$*, patch generation involves creating a candidate patch* $\mathcal{CP}$ *that transforms the buggy program* $\mathcal{P}$ *into a new program* $\mathcal{P}'$*.*

Definition 2.1.4 (Patch Validation) *For a buggy program* $(\mathcal{P}, \mathcal{O})$ *and its associated candidate patch* $(\mathcal{CP})$*, patch validation is the process of assessing whether the patch* $(\mathcal{CP})$ *is able to transform the original program* $(\mathcal{P})$ *into a new program* $(\mathcal{P}')$ *that successfully complies with the explicit oracle* $(\mathcal{O})$*.*

2.1.1 Heuristic-Based Repair

Heuristic-based repair [50, 36, 51, 25, 52, 53, 16, 54, 38, 41] constitute a collection of notable APR approaches that employ heuristics or predefined rules to guide the search for potential bug fixes in software programs. This method leverages search-based algorithms, genetic programming, and evolutionary algorithms for an iterative exploration of the program modification space. It is commonly described as a generate-and-validate process, where potential patches for suspicious bug locations are generated and then validated against predefined oracles, such as test suites, to evaluate their effectiveness in resolving the identified bugs. Genetic Programming (GP) techniques, including GenProg [51], utilize evolutionary algorithms to evolve a population of candidate patches over time. Search-based repair approaches, such as RSRepair [52], SPR [53], and HDRepair [16], incorporate meta-heuristic search

algorithms to traverse the patch search space. Template-based repair methods, exemplified by tools like TBar [50] and PAR [54], generate candidate patches by drawing upon predefined templates originating from common bug-fixing patterns. Heuristic-based repair excels in efficiently exploring vast solution spaces and producing diverse patches; however, it may not offer formal guarantees on patch correctness. This limitation can potentially lead to overfitting patches and computationally intensive search processes.

2.1.2 Constraint-Based Repair

In contrast to heuristic-based repair, constraint-based repair [55, 56, 57, 58, 59, 20, 60] represents another category of popular APR approaches that utilize formal methods to model and address the program repair problem as a constraint satisfaction or optimization problem. Upon identifying the faulty code, this method transforms the extracted specifications and faulty code into a set of constraints representing the desired program behavior, employing techniques such as symbolic execution, abstract interpretation, and model checking. Subsequently, it leverages constraint solvers like Satisfiability Modulo Theories (SMT) or constraint logic programming to synthesize a patch that satisfies these constraints for the faulty code. Lastly, it verifies the generated patch's effectiveness by executing the test oracle. For example, SemFix [20] and KLEE [60] use symbolic execution, a technique that analyzes programs by substituting input values with symbolic expressions, to encompass all possible values. This procedure generates constraints on these expressions, which are then solved by a constraint solver to determine concrete input values that satisfy the constraints. Concolic repair is a program analysis technique combining concrete execution (running a program with specific input values) and symbolic execution (replacing input values with symbolic expressions) to explore various paths within a program. By integrating both techniques, concolic repair approaches like CPR [56] can generate path constraints and uncover feasible inputs to cover multiple execution paths, assisting in identifying and resolving software bugs. Constraint-based repair, emphasizing system consistency, efficiently addresses constraint violations and enhances the program's reliability and robustness. However, this approach may be limited by the scalability and complexity of constraint solvers, potentially reducing its scalability to larger and more intricate programs. Furthermore, similar to search-based APR tools, constraint-based methods may also suffer from patch overfitting due to their reliance on weak test cases for correctness evaluation.

2.1.3 Learning-Based Repair

Recent advancement in APR is Learning-based repair [61, 62, 63, 15, 14, 12, 38, 64, 65], which leverages machine learning techniques such as deep learning, reinforcement learning, and probabilistic modeling to identify patterns and generate patches based on existing code or previous repair instances. This approach aims to capitalize on the wealth of knowledge available in extensive code repositories to train models that effectively synthesize bug-fixing patches with minimal human intervention. In deep learning-based repair, tools like DeepFix [12] and RewardRepair [15] employ neural machine translation (NMT) with sequence-to-sequence neural networks based on encoder-decoder architectures to automatically learn the likelihood of correctness for candidate patches and rank them accordingly, facilitating the discovery of the correct patch. A significant breakthrough in learning-based repair is the use of

LLMs. Codex [66, 62] (the model behind Github Copilot [67]), a state-of-the-art language model developed by OpenAI, has demonstrated considerable potential in the field of automated program repair. Built on the GPT-3 architecture, Codex has been fine-tuned on a diverse range of programming languages and codebases, allowing it to generate higher-quality bug fixes. ChatGPT [61, 68, 69, 70], a recently introduced LLM, has garnered significant interest due to its reported exceptional capabilities in automatically repairing software. It utilizes advanced supervised instruction fine-tuning techniques and RLHF to adapt more effectively to specific tasks or domains, substantially outperforming previous LLMs. The exhibited results suggest that LLM-driven software engineering, such as LLM-based program repair, holds a promising future. While learning-based repair can adapt to new domains and contexts, as it depends on learning from past examples rather than predefined heuristics or extracted constraints, it may be susceptible to overfitting issues. This is because predictions are made based on the training dataset, which lacks new and unseen knowledge, relying on historical memory rather than formal logical reasoning.

2.2 Patch Overfitting

APR methodologies present an opportunity for developers to streamline debugging through the generation of patches, which enable a flawed program to satisfy a given oracle, such as a test suite. These patches are referred to as *plausible patches*. In practice, through manual inspection, the test-passing patches are divided into two categories: correct and overfitting. The correct patches implement the expected behavior and effectively address the defects in realistic scenarios, while the overfitting patches do not. We, in this dissertation, focus on the challenge of patch overfitting. Patch overfitting in APR occurs when the generated patch repairs the given bug in the context of the specific oracle but may not be a correct or general fix for the underlying problem in the code. This problem leads to a diminished rate of defect repair in real-world software applications. It is to be noted that the concept of patch overfitting, also known as test overfitting, has been presented in the literature from various perspectives. However, these differing interpretations of the causes of patch overfitting may confuse associated readers, especially those new to the field. To alleviate this, we classify patch overfitting into three categories based on its causes, as follows:

- **patch overfits to weak oracle.** During the validation process, patches are produced to satisfy the test oracle, which sets the correctness criteria. However, given the oracle's inherent weakness and inability to cover all potential facets of the bug, the patches, though compliant with the current oracle, may still be incorrect. It indicates that the incorrect patches fail to generalize. As a result, APR tools tend to generate patches that overfit to the test oracle (*e.g.,* test suites) [26].
- **patch overfits to incomplete specifications.** In the generation process, the APR execution is guided by program specifications, which may use test suites, constraints, or static analyzers. These specifications typically depict the desired program behaviors that the APR tools strive to satisfy with. Unfortunately, only part of the required behaviors is explicitly specified. Therefore, the APR-generated patches are prone to overfit to incomplete specifications [71, 11].
- **patch overfits to training corpus.** Recently, learning-based techniques (*e.g.,* LLMs based repair) have progressively become central to APR approaches. We thus propose a new perspective to introduce patch overfitting in the learning-based APR techniques. In the terminology of machine learning, "overfitting" typically denotes a model's propensity to fit too closely to the training data, consequently failing to generalize its performance to unseen or novel data. This issue similarly pervades learning-based APR tools. These tools commonly rely on a substantial training corpus—comprising codes or patches—to construct sequence-to-sequence models for generating code to address faulty programs. However, the synthesised patches may overfit to the patterns within the training corpus, thereby failing to effectively repair new and unseen bugs.

In the preceding discussion, we present the causes of patch overfitting. Concurrently, existing literature [1] proposed the classification of overfitting patches based on their impact on the original program. This classification operates under the assumption of input space ($\mathcal{I}$) of a program ($\mathcal{P}$) in the context of object-oriented programs. An input point consists of one or more objects interacting through method calls. In typical repair scenarios, a bug only impacts a portion of the input domain,

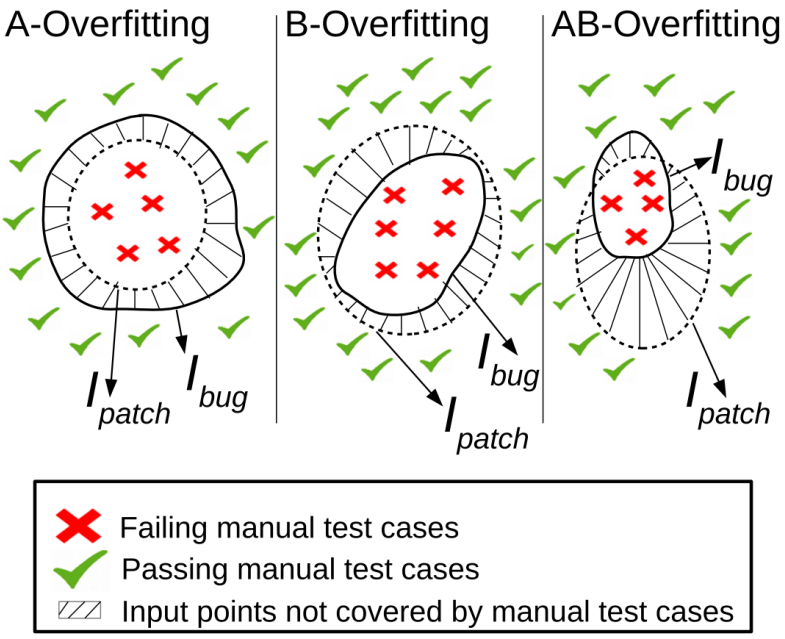


Figure 2.2: The classification of the overfitting patch [1].

termed the "buggy input domain" ($\mathcal{I}_{bug}$). The remaining input domain, where program behaviors are correct, is labeled as $\mathcal{I}_{correct}$. By definition, an automatic program repair technique generates a patch that alters the behaviors of some input domain ($\mathcal{I}_{patch}$). Patches can correctly or incorrectly change the original buggy behaviors within $\mathcal{I}_{bug}$, denoted as $\mathcal{I}_{patch1}$ and $\mathcal{I}_{patch=}$, respectively. Modifications to the behaviors of input points within $\mathcal{I}_{correct}$ can result in correct behaviors becoming incorrect, denoted as $\mathcal{I}_{patch0}$. The total affected input domain ($\mathcal{I}_{patch}$) is the union of $\mathcal{I}_{patch=}$, $\mathcal{I}_{patch1}$, and $\mathcal{I}_{patch0}$.

For a specified faulty program, an ideal patch should rectify all input points within $\mathcal{I}_{bug}$, while concurrently ensuring that no input points within $\mathcal{I}_{correct}$ are compromised. Nevertheless, due to the reasons discussed above, the patches that are produced are often imperfect and typically exhibit overfitting. As per their interaction with the input domains $\mathcal{I}_{bug}$ and $\mathcal{I}_{correct}$, two types of overfitting problems have been discussed in the literature [1, 72].

Incomplete Fixing: The generated patch only repairs some, but not all, input points within $\mathcal{I}_{bug}$. In other words, $\mathcal{I}_{patch1}$ is a proper subset of $\mathcal{I}_{bug}$ ($\mathcal{I}_{patch1} \subset \mathcal{I}_{bug}$).

Regression Introduction: The generated patch broke some input points within $\mathcal{I}_{correct}$. In other words, $\mathcal{I}_{patch0}$ is not an empty set ($\mathcal{I}_{patch0} \neq \emptyset$).

Building on these two distinct types of overfitting problems, three different categories of overfitting patches have been defined. Note that we assume the overfitting patches have passed all inner test oracle.

A-Overfitting Patch: This type of overfitting patch exhibits the issue of incomplete fixing, signified by $\mathcal{I}_{patch1} \subset \mathcal{I}_{bug} \wedge \mathcal{I}_{patch0} = \emptyset$. This kind of overfitting patch can be considered a "partial patch".

B-Overfitting Patch: This type of overfitting patch is specifically characterized by the regression overfitting issue ($\mathcal{I}_{patch1} = \mathcal{I}_{bug} \wedge \mathcal{I}_{patch0} \neq \emptyset$). It's crucial to note that this type of overfitting patch is capable of correctly rectifying all input points within the buggy input domain ($\mathcal{I}_{bug}$), yet simultaneously, it breaks certain previously correct behaviors of the program.

AB-Overfitting Patch: This type of overfitting patch simultaneously exhibits

both incomplete fixing and regression introduction problems ($\mathcal{I}_{patch1} \subset \mathcal{I}_{bug} \wedge \mathcal{I}_{patch0} \neq \emptyset$). This particular type of overfitting patch rectifies some, but not all, input points within the buggy input domain, $\mathcal{I}_{bug}$, while also introducing certain regressions.

Figure 2.2 illustrates the three distinct types of overfitting patches.

By understanding the causes of patch overfitting and the different types of overfitting patches generated, the APR community can strategize and devise approaches to effectively tackle the patch overfitting problem.

2.2.1 Empirical Studies of Patch Overfitting

The patch overfitting problem has held the attention of the APR community since the analysis of patch plausibility and correctness was conducted [25]. Subsequent research has taken diverse perspectives in conducting numerous empirical studies on this problem. In this section, we review these works with the intention of both enhancing understanding of the problem and providing inspiration for future research endeavors.

Severity of patch overfitting. Initial researchers recognized the problem of patch overfitting and consequently carried out empirical studies to assess the severity of this issue [25, 26, 73]. Qi et al.[25] observed that the majority of patches produced by GenProg[74], RSRepair [52], and AE [75] systems are semantically equivalent to a single modification—eliminating functionality while still producing accurate outputs for the inputs presented in the test suites. However, the patches are still incorrect as they obviously break the untested functionality (*i.e.,* regression introduction). The quality of generated patches also drew the interest of Smith et al. [26]. They employed independent test cases to evaluate plausible patches that had already passed all training test cases used during the repair process. The insights revealed that patches tend to overfit the training test suite, often breaking undertested functionality and thereby resulting in incorrect outcomes. Their analysis of the correlation between training suite coverage and patch overfitting demonstrated that test suites with higher coverage tend to produce higher quality patches—i.e., less overfitting. This insight serves as a stepping stone for future research on augmenting test cases to mitigate patch overfitting [34, 35, 1, 37]. Furthermore, in order to assess whether a patch could make the program worse, they compared the test passing rate before and after patching the program. Their results indicated that TrpAutoRepair [76] patches are more prone to break original undertested functionality, while GenProg [74] patches exhibited lesser overfitting. In addition, the patches perform no worse than novice developers who also overfit to training test cases.

Patch overfitting in different APR techniques. Inspired by prior works, researchers started exploring the prevalence of patch overfitting in specific APR techniques [77, 78, 79, 80]. Long et al.[77] offered a systematic analysis of the crucial characteristics of patch search spaces examined by two APR systems—SPR [53] and Prophet [38]. This kind of system applies transformations to suspicious statements to generate patches. Concentrating on the density of correct and plausible patches, they discovered that the explored search spaces often contained hundreds to thousands of times more overfitting patches than correct ones. Additionally, larger search spaces result in fewer correct patches, attributing this to the increased presence of plausible patches that obstructed the identification of correct patches. Consequently, exploiting information beyond the test suite to filter out plausible but overfitting patches can potentially facilitate the isolation of correct patches. This insight

prompted subsequent researchers to successfully harness relevant information sources, such as bug reports [81, 82] and open-source code repositories [38, 41], to mitigate patch overfitting. While previous studies have explored patch overfitting in heuristic-based APR tools, semantic-based APR represents a different methodological branch. This technique synthesizes program repair that complies with constraints extracted from symbolic execution and test suites. Le et al. [78] examined semantic-based APR techniques, systematically verifying, characterizing, and comprehending the characteristics of overfitting within these under the IntroClass [83] and Codeflaws [84] benchmarks. Following this, recent methods [56] are devised to tackle the unique overfitting issues in these techniques. Dynamic APR approaches (*i.e.,* require dynamic execution of the program) typically achieve a high repair rate. Nilizadeh et al. [79] endeavored to evaluate patch overfitting within dynamic APR. To validate patch correctness, they proposed the use of formal methods (specification and verification) as an independent standard. The experimental evaluation confirmed the existence of overfitting within dynamic APR tools, despite the fact that around 59% of patched programs were correct.

Difference between overfitting and correct patches. To better guide APR approaches towards generating correct patches, researchers proposed examining the differences between overfitting patches and correct ones [85, 48, 48, 86]. Wang et al. [85] carried out an empirical study on APR-generated and developer-written patches. From a statistical standpoint, they discovered that 25% of the correct patches (primarily Same Location Different Modification (SLDM) and Different Location Different Modification (DLDM)) produced by APR are syntactically different from those written by developers, whereas the majority were identical. This finding implies that future APR approaches might not necessarily need to generate patches that are syntactically identical, but instead can concentrate on producing semantically identical ones. Yang et al. [48] investigated the different runtime behaviors triggered by correct patches and plausible yet overfitting patches. One intuitive assumption about patch correctness is that a correct patch may have a greater impact on program behavior when executed by failing test cases, and a lesser impact when executed by passing ones. Overfitting patches, conversely, exhibit the opposite effect. Based on this assumption, they proposed using Daikon [87] to infer program invariants that represent runtime behavior, thereby quantifying the change in runtime behavior between the buggy program and the correctly (or incorrectly) patched program. Experimental results performed on the Defect4J benchmark [88] revealed that out of 73 correct patches, 72 influenced the invariants generated from failing test cases, and 43 also impacted the invariants generated from passing test cases. Finally, they verified the presence of a difference in the invariants affected by correct and overfitting patches, substantiating the feasibility of using program invariants to distinguish between correct and overfitting patches. Bennett et al.[86] endeavored to analyze several features contributing to the inaccuracies in plausible patches by comparing them to developer-written patches. First, they examined patch size, building upon the existing idea of assessing defect complexity based on the number of change actions needed for a correct patch [30]. Their analysis suggested that APR tools struggle with handling multiple actions simultaneously. In other words, patches that only modify a single line are more prone to overfitting. This finding indicates a need for the APR community to improve its ability to generate multi-line patches. Secondly, they evaluated the impact of repair actions and discovered that overfitting patches

more frequently occur in defects requiring the addition of a method call or a variable. This insight provides direction for future APR tools to prioritize the synthesis of new code, particularly concerning method calls and variables. Finally, they investigated repair patterns. Defects necessitating the addition or removal of a try/catch block or loop exhibited a strong correlation with overfitting patches. In conclusion, their research scrutinized the characteristics of overfitting patches in relation to unfixed defects, thereby shedding light on the limitations currently present in APR tools.

Benchmarks and metrics. Contrary to the previously mentioned studies, some researchers have examined patch overfitting through the lens of benchmarks and metrics, such as test suites [89, 90, 91, 29, 92, 71, 86, 93]. Defects4J [88], a large dataset for software engineering, offers 835 reproducible bugs from a diverse array of real-world, open-source Java projects, along with supporting infrastructure for multiple research tasks, including program repair. To examine overfitting across different programming languages or program repair techniques, Martinez et al.[90] carried out a large-scale manual assessment on the 84 patches generated for Java projects in the Defects4J dataset, contrasting them with the C projects in Qi et al. [25]. Their findings are consistent with earlier work [25, 26], confirming that most generated patches tend to overfit to incomplete test data. They validated the prevalence of overfitting for Nopol [57] and discovered that they could not validate the accuracy of 12 patches due to the absence of expert domain knowledge. These results underscore the necessity for efficient patch correctness assessment techniques. Meanwhile, Jiang et al. [91] manually analyzed Defects4J defects to provide a human-centric explanation of failed overfitting patches. They successfully generated plausible but overfitting patches for 6 out of 9 defects that could not be fixed. Upon further inspection, they discovered that the test suites lacked sufficient information to fully elucidate all aspects of the defects, resulting in incomplete and overfitting patches. Their analysis also illuminated the potential for test case augmentation to improve patch accuracy. Additionally, they investigated patch generation strategies, suggesting that the strategy of returning expected output with a return statement heavily relies on the developer's experience to circumvent overfitting. Overall, despite the problem of patch overfitting, existing test suites aided developers in resolving a significant portion of defects on Defects4J, with 82% being successfully addressed.

QuixBugs [92], a widely-used benchmark, consists of 40 one-line defects equipped with both failing and passing test cases. In an extensive study conducted by Ye et al. [94], the repair status of bugs within QuixBugs was closely examined. It was revealed that out of 40, 24 buggy programs had not been repaired by any of the ten repair tools investigated by the authors. Further examination pointed to the fact that the overfitting patches that were generated either failed to implement the mutation of the operator or lacked the essential components necessary for the repair process. Moreover, when considering patches, 53% (180 out of 338) of the unique (deduplicated) patches created for QuixBugs were overfitting, thereby negatively impacting the efficiency of APR tools and eroding the trust of engineers. Finally, they evaluated the effectiveness of several automatic patch assessment techniques on QuixBugs. Test generation-based approaches, $RGT_{Evosuite}$ [27] and $RGT_{InputSampling}$ [95], achieved accuracies of 98% and 80%, respectively. However, the invariant detection-based approach, $GT_{Invariants}$ [48], underperformed with an accuracy of 58%.

Zemin et al.[89] assessed the suitability of test cases as proxy acceptance metrics for automated program repair. They investigated the performance of APR tools like

Nopol [57], GenProg [74], Angelix [58], and AutoFix [96] in conjunction with various sizes of test suites. Their experimental findings suggested that larger test case sizes help reduce overfitting patches for Nopol, a synthesis-based technique, but not for other tools. Liu et al. [29] proposed the creation of bias-limited metrics to evaluate the performance of APR systems from different perspectives. This was prompted by the shift away from plausible patches as the primary metrics due to the prohibitive cost of assessing correct patches. They calculated the number of invalid patches that must be verified before finding a plausible patch, which could serve as a platform-independent and reliable metric for comparing the effectiveness of state-of-the-art approaches. Additionally, by evaluating the applicability of state-of-the-art solutions, they noted the risk of current APR systems or generated patches overfitting to used benchmarks. This insight encourages researchers to enhance the ability of APR tools to generate more generalized patches.

2.2.2 Addressing Before Patch Generation

Fault Localization

Fault localization (FL), the first step in APR, aims to identify potential locations of faults in code space for further repair. While it is often treated as a separate process from patch generation, the correctness of generated patches is heavily dependent on the FL [97, 98, 91, 99, 100, 101, 102]. Therefore, researchers began to attempt to investigate the early stage (FL) of the APR pipeline to prevent the subsequent emergence of overfitting patches. Since our topic is related to patch overfitting, we only present the works that aim to address patch overfitting by explicitly correlating FL and patch correctness instead of the ones that only focus on improving FL itself.

I. Heuristic focused.

Based on heuristic, researchers propose integrating FL techniques to facilitate the generation of high-quality patches [103, 104]. Mechtaev *et al.* [103] presented a novel approach that aims to find simpler patches than state of the art APR tools as well as minimize the impact on the original program structure. The motivation of their work is that simple patches are often preferred by researchers and are more likely to be accepted [105, 106]. The existing APR tools are prone to produce complex patches as their fault localization was an independent process that is conducted before patch generation, and thus does not include the computation of the simplicity of repairs. To obtain simple patches, the authors, for the first time, integrated fault localization and patch generation into one process, making the selection decision of fault location considering the simplicity of repair patches. The experimental results showed that DirectFix, a tool implemented by authors, generated simpler patches than SemFix [20], causing substantially less frequent program regression errors (*i.e.,* patch overfitting). Xu et al. [104] hypothesized that candidate patches, which successfully pass part of the test suites that the original program with bugs failed, are likely to be closer to the correct solution. They thus proposed retrospective fault localization, which reuses instead of just discarding, the candidate patches that fail on validation to enhance the quality of patches when integrating with APR techniques.

II. Precision focused.

Other researchers attempted to improve the accuracy of FL for enhancing the generation of correct fixes [107, 108, 109, 50, 99, 110]. For instance, based on the

rich state-based abstraction of the program's behavior, JAID [108, 109] improves the accuracy of fault localization and the generation of state-modifying patches. Correspondingly, JAID generates accurate patches based on the semantic analysis that determines how to alter the object state to prevent failure. Koyuncu *et al.* [107] explored bug report information-driven program repair built on Information Retrieval (IR)-based fault localization. Experimental analysis from the Defects4J dataset indicates that fault localization based on IR tends to produce fewer overfitted patches compared to Spectrum-based fault localization methods when suggesting code modifications. Their proposed APR system, iFixR, is capable of generating and prioritizing more correct patch recommendations for a diverse range of user-reported bugs. Afzal *et al.* [99] developed a semi-automated program repair tool SOSRepair$^{\oplus}$, which is provided with manually-specified candidate fault locations. Compared with SOSRepair using spectrum-based fault localization, the results show that SOSRepair$^{\oplus}$ produced twice as many high-quality patches (16 versus 9) that pass all held-out tests. Motwani *et al.* [110] built a novel FL technique SBIR that combines a spectrum-based (SBFL) technique and Blues, an unsupervised FL strategy at the statement level, which utilizes bug reports and tests. SBIR has demonstrated its ability to rank buggy statements as more suspicious than those that are not faulty, thereby significantly minimizing repair failures attributed to localization inaccuracies and improving patch quality for FL-sensitive APR tools. Xu *et al.* [102] employed value-flow analysis to rank suspicious buggy statements for their repair operation. By accurately selecting a repair location and implementing the correct operation, they identified potential Null Pointer Exceptions (NPEs) that were not triggered by the provided test suite, resulting in the generation of correct patches.

While current approaches have made strides in utilizing FL techniques to mitigate patch overfitting, they come with their own sets of challenges. Heuristic-based methods, though designed for the simplicity of the patch, can produce patches that are misleadingly elegant yet incorrect—demonstrating that simplicity is not synonymous with accuracy. Conversely, methods focused on precision often incur high computational costs and are still susceptible to generating a multitude of overfitting patches, despite accurate FL. This leaves users with the tough task of distinguishing between correct and incorrect patches. Moreover, these approaches generally fail to capture the semantics of the underlying fault, a crucial element for aligning with the behavior of a genuinely correct patch. As a result, merely fine-tuning FL is insufficient for capturing the semantic nuances of correct patches in the context of patch overfitting. Our research aims to directly tackle this issue by focusing on efficiently evaluating patch correctness through a nuanced understanding of their semantic behavior, particularly in relation to a bug.

Patch Generation

Researchers employ diverse techniques to construct methodologies for auto-generating patches for programs with bugs. However, initial APR tools, owing to their disregard for the weakness of the test oracle, tend to generate overfitting patches [25]. As the APR community has recognized the significance of patch overfitting, efforts have been directed towards optimizing patch generation tools. The objective is to enhance the quality of the patches produced, thus avoiding the direct generation of overfitting patches.

I. Heuristic based.

By enhancing the search optimization process for specifications or codes, researchers address overfitting in heuristic-based APR [111, 103, 112, 41, 113, 16, 114]. As previously discussed, there exists the risk of patches overfitting to incomplete specifications. Some research suggests driving APR through historical open-source codes. Ke *et al.* [111] argued that human-written code often embodies a subtle understanding of correctness that may not be wholly represented by partial test suites. Therefore, reusing such code has a higher probability of yielding patches that coincide with the desired functionality (*i.e.,* unwritten specification). They proposed the construction of input-output profiles to guide a code semantic search for larger blocks of human-written code, such as entire method bodies, which are more likely to satisfy the unwritten specification of correct program behavior compared to randomly generated smaller fixes. Likewise, Le *et al.* [16] extracted fixing patches in revision control systems of previous software to inform the construction of candidate patches. Besides, the fixed history is used to help assess the quality and fitness of the patches in the selection phase. On the other hand, van Tonder *et al.* [112] observed that prior strategies often overfit to the available dynamic specifications of desired behavior, such as test suites. Consequently, they introduced FootPatch, a static approach that takes into account the logical encoding and semantic implications of searching for the existing program fragments via a repair query to satisfy repair specifications. They argued that their approach, which places an explicit emphasis on editing semantic effects, prevents patch overfitting.

Certain studies focus on devising methods to give priority to correct fixing ingredients within a pool of plausible ones. For instance, to pursue high-quality patches, Le *et al.* [41] formulated a two-phase approach. Firstly, they utilize dynamic symbolic execution on the provided test cases to automatically derive examples that act as a specification of the correct behavior. Secondly, they employ a domain-specific language (DSL) to systematically tailor and restrict the solution search space. An enumeration-based method is then used for efficiently traversing this space to synthesize solutions with broad scalability. Finally, they proposed a patch quality ranking strategy based on syntactic and semantic distance.

Navigating the fine granularity search space, which has a higher likelihood of containing correct fixing components, also brings with it the challenge of a more complex search process. To address this, Wen *et al.* [46] developed a context-aware prioritization method for mutation operators to constrict the search space. Furthermore, they enhanced the efficiency of the search process by prioritizing more likely correct patches using three novel models trained on context information of AST nodes. Their evaluations illustrate that their approach is capable of generating plausible patches for 25 bugs of the Defects4J benchmark, with 21 of them being correct, thereby achieving a high precision rate of 84%.

II. Constraint based.

By refining the constraints employed to guide the synthesis of patches, researchers address overfitting in constraint-based APR [44, 115, 116, 1, 117]. Directly returning the test oracle of the failed test in a repair can be regarded as overfitting because a test oracle is usually designed for a specific test input instead of generalized other inputs outside. Xiong *et al.* [44] revisited the repair operation by synthesizing the precise conditions (*i.e.,* specifications) at the buggy location. The precise conditions enable ACS, a Java program repair tool developed by them, to generate precise patches, that have a relatively high probability of being correct. Mechtaev *et al.* [115] investigated

the potential to infer missing specifications of intended behaviors from correct reference implementations. A reference program can be regarded as an alternate implementation of the same functionality and is commonly available for software such as web servers, libraries, and database management systems. Specifically, given a reference program and an input condition, they deduce a symbolic summary of the path conditions and symbolic output states, which are subsequently used as a specification. Leveraging these inferred specifications, their proposed method, SemGraft, generated a larger quantity of repairs equivalent to developer patches. This suggests that their approach provides additional correctness assurances on the generated patches and is able to scale effectively to real-world programs, including those such as GNU Coreutils [118] and Busybox [119].

Several tools, like SemFix [20] and Nopol [57], rely on constraints extracted from human-written test suites, which can often be incomplete. To address this, Yu *et al.* [1] proposed the enhancement of these test suites using EvoSuite to establish more robust associated constraints. On the other hand, Shariffdeen *et al.* [56] rely on user-provided specifications, *e.g.,* a constraint on some specific behavior. They employed concolic path execution to concurrently explore both the input space and the patch space. Meanwhile, they reduced the pool of patch candidates by excluding any patches that failed to meet user-provided specifications. Additionally, they deprioritized patches that significantly altered the behavior of the original program. Gao *et al.* [117] extracted constraints from the symbolic execution of the partial path of the program to repair security vulnerabilities. The constraints are generalizable and can provide additional guarantees in generating crash-free patches.

III. Learning based.

By utilising or optimizing machine learning models, researchers address overfitting in learning-based APR [38, 15]. Prophet [38] conducted the first attempt to develop an APR system using machine learning methodologies that harness the properties of correct codes. The approach taken by this system involves the use of transformation schemas, staged program repair, and condition synthesis [53] in order to produce a search space filled with partially instantiated candidate patches carrying key variables. Then, Prophet employs a machine learning model, trained on program value features, to attribute and rank probabilities to patches in descending order of their correctness score. The results demonstrate that out of 19 defects successfully repaired by Prophet and other state-of-the-art systems, Prophet managed to rank the correct patch first in the list of candidate patches for 15 of them, whereas the other systems achieved this for only 11. Previous NMT-based program repair studies have solely optimized a loss function that is purely syntactic, focusing only on characters and tokens, without integrating any program-specific semantics during the optimization of neural network weights. Ye *et al.* [15] proposed RewardRepair to avoid overfitting in NMT-based program repair. This method involves optimizing a loss function, incorporating the execution information from new test cases, generated based on the ground truth patched program. They introduced a regression discriminator which delineates behavior beyond the range of developer-written tests, thus rewarding the network to adjust weights for the production of non-overfitting patches.

Despite advances in APR methods to minimize patch overfitting, each strategy has its drawbacks. Heuristic-based approaches excel in generating simpler patches but may compromise on correctness and generalizability. Constraint-based methods often necessitate exhaustive specifications or depend on potentially incomplete test suites,

risking overfitting. Learning-based models, such as machine learning and neural networks, might produce patches that are syntactically correct but semantically flawed due to their limitations in understanding code semantics. Extending search constraints can reduce overfitting but also limits the discovery of truly correct solutions, creating a trade-off between avoiding overfitting and finding a reliable, generalizable patch. In contrast, our research maintains an unrestricted patch space, using statistical techniques to efficiently identify correct patches post-generation. Additionally, we introduce hypotheses to capture the semantic relationship between correct patches and the underlying bugs.

2.2.3 Addressing After Patch Generation

Patch Validation

Once the patches have been produced, they undergo a validation process against a test oracle to evaluate their correctness. While not perfect, test oracle can reveal potential defects or vulnerabilities in patched programs by acting as approximations of the program's intended behavior [26]. Researchers proposed enhancing the test oracle to uncover defects in the patched program with the goal of filtering out overfitting patches in advance.

I. Reference based.

The human-written patch, which encodes the correct program behavior, is regarded as an oracle. Some research proposed to generate new test cases through the oracle [35, 27, 78, 1]. Xin *et al.* [35] introduced DiffTGen, a methodology developed to identify test-suite-overfitted patches. The approach begins by formulating new test inputs, highlighting semantic variances between the original buggy program and the human-written corrected version. These semantic differences guide the subsequent testing of the patched program, and yield additional test cases. Finally, the incorporated test cases not only enhance its robustness but also prevent the future generation of similarly overfitting patches. An evaluation of DiffTGen was conducted on 89 patches produced by four APR tools for Java, 79 of which were suspected to be overfitting and incorrect. The results showed that DiffTGen successfully identified 39 (49.4%) overfitting patches, generating the associated test cases. By implementing Random Testing based on Ground Truth (RGT), Ye *et al.* [27] generated additional test cases to distinguish program behavior between ground truth patches and program repair patches, with the goal of filtering out overfitting patches. This strategy led to a 190% performance improvement for DiffTGen on the same benchmark. However, these approaches assume the presence of a reference patch (expected behaivor), which is often not accessible in real-world scenarios.

II. Non-Reference based.

To scale to realistic applications, researchers have proposed approaches that operate without such a reference [34, 36, 37, 2, 120, 121]. To that end, Yang *et al.* [34] suggested creating new test cases by implementing a fuzz strategy on the inputs of existing test cases. Furthermore, they supplemented the weak test oracle with crash and memory-safety oracles to improve the validity checking of the generated patches, *e.g.,* whether they contained a memory-related bug. Similarly, Gao *et al.* [36] introduced crash-freedom as a new oracle to eliminate crashing plausible patches. Their goal was to narrow down the realm of correct patches by distinguishing crash-free ones from the entire plausible patch set. More specifically, they prioritized new

tests generated by a grey-box fuzzing strategy that covered functionalities differing across plausible patch candidates. Such a test suite is more likely to distinguish between crash-free and crashing patches. However, these methods do not tackle the issue associated with the absence of expected behavior in the programs being tested. Smith *et al.* [37] put forward a method for learning an automatic oracle that can predict labels for newly generated test cases, drawing on program inputs and actual outputs. This strategy leverages these generated labeled test cases to enrich the original test suite, aiming to efficiently eliminate overfitting patches. Gissurarson *et al.* [121] introduced properties in addition to unit tests to address patch overfitting. Although the number of programs that can be patched has not increased, the solutions were found to be less overfit.

While patch validation strategies help mitigate overfitting risks, they have inherent limitations. Reference-based approaches assume the presence of a human-generated patch as an oracle, an assumption often impractical in real-world contexts. On the other hand, non-reference-based methods are more scalable but rely on heuristics such as fuzzing or crash-freedom, potentially overlooking domain-specific issues and nuanced bug semantics. Our research uses artificial intelligence heuristically to capture the essential semantics of a correct patch, bypassing the need for a strong test oracle.

Patch Correctness

Although test oracle ensures a certain level of quality for patches generated during patch validation, the correctness of these patches remains subject to further scrutiny. Upon the large number of plausible patches [28] and developers' willingness to review a few of them [31], researchers proposed various approaches for automatically assessing patch correctness. Technically, utilizing diverse sources of information related to patch correctness, they aim to heuristically prioritize or identify the correct patches while filtering out overfitting ones. By automating the evaluation of patch correctness, the research is aimed at reducing the human burden involved in reviewing plausible patches generated by patch generation tools.

I. Patch focused.

Given that the patch is the subject of patch correctness assessment, lots of approaches-mostly static-have been proposed to exploit patch-focused information [40, 116, 41, 44, 46, 47, 122, 123, 123, 48, 124, 47, 124, 125, 126, 49, 27, 127, 128]. To guarantee the correctness of the generated patches, researchers adopted author annotation, *i.e.,* manually evaluate whether a patch is semantically equivalent to the developer-written patch [129, 17, 130, 131, 132]. Although this method is effective, it places a substantial demand on human effort when a large number of plausible patches need to be validated. Moreover, it requires the availability of a developer-written patch, a pre-condition that possibly is not feasible in real-world bug-fixing scenarios.

Tan *et al.* [40] proposed the construction of a set of anti-patterns, with the premise that patches infringing these rules should be classified as incorrect. More specifically, they undertook a manual analysis of both APR-generated and human-written patches, from which they deduced a number of prohibited transformations. Utilizing these transformations, the authors devised related anti-patterns that, when incorporated into the search-based APR process, serve to eliminate overfitting patches. There is a widely-used hypothesis that small code change (patch) is more likely to be correct [42, 43, 45, 18]. Based on the hypothesis, a large number of studies

investigated the similarity between buggy and patched codes to prioritize or identify correct patches [116, 41, 44, 46, 47, 123, 123, 48, 124]. Le *et al.* [41] and Wen *et al.* [46] integrated a patch prioritization component into their repair tools to select more correct patches by comparing the code changes in different measurements such as AST differencing, Cosine similarity, etc. Cashin *et al.* [47] clustered semantically similar patches by distinguishing the patch differences through formal invariants, reducing the number of patches to be reviewed by developers. Ghanbari *et al.* [124] employed a quantitative method to examine the syntactic and semantic similarity between a patched program and its original version based on the production and test codes. Their proposed technique, Shibboleth, initially determines token-level syntactic similarity by constructing vectors based on the frequency of tokens across all patched methods, which is then measured by comparing the cosine similarity. Shibboleth [124] leverages dynamic execution to calculate the cosine similarity of what is referred to as the Statement-Count Spectra for the program both pre/post-patching. This combined approach allows for a comprehensive and nuanced analysis of program modifications.

Machine learning and pre-trained language models have been employed to learn and extract information from codes or patches, thereby enabling the reasoning of patch correctness [125, 126, 49, 27]. Csuvik *et al.* [125] use representation learning embedding models Doc2vec and Bert to encode patch codes to filter out the patches that make larger changes. Following up on it, Lin *et al.* [126] also utilized the power of representation learning techniques. Instead of encoding the code tokens of the patch, they proposed to capture more contextual and structural information by leveraging the AST path of the buggy statements, patched statements and unchanged context. There have been other works that investigated the naturalness difference between buggy code and correct code [133, 38]. Inspired by these studies, Kang *et al.* [49] suggested the use of language models to evaluate the naturalness of a patch, thereby reasoning about its correctness. Experimental results have shown that the prioritization of patch naturalness by language models often surpasses the default prioritization strategies employed by five APR tools. Ye *et al.* [27] developed ODS, a static supervised learning based system to classify correct and overfitting patches. ODS leverages diverse syntactic code features such as code description, repair pattern, and contextual information. These features are extracted from a large-scale labeled patch dataset, with their characteristics captured at the granularity of the AST.

While APR tools aim to generate patches aligning with the developer's goals, the incompleteness of the associated specifications often inhibits these generated patches from achieving their intention. Phung *et al.* [127] conducted an analysis extracting the method names from faulty code snippets, which they interpreted as indicative of the original developer's intention. Following this, they employed Code2Vec to extract the semantic meaning - essentially, the actual behavior - embedded within the patch code. In the final step, they computed a similarity score between the extracted intended and actual behaviors, using this metric to identify patches with low similarity scores as likely incorrect solutions. Unlike the previous approaches, Gao *et al.* [39] propose a human-in-the-loop patch evaluation scenario in which developers review question-answer sets to select the correct patches. Their approach involves automatically translating patch semantics into *what* and *how* questions, which respectively represent the desired program behaviors and the necessary program modifications to achieve these behaviors. Developers then pose *what* questions to

choose answers from a range of Angelic values [134], and *how* questions to select the patches. This interactive method of patch generation and recommendation simplifies the process for developers, enabling them to identify the correct patch without the need for detailed comprehension and analysis of the semantic changes across a multitude of plausible patches.

II. Bug focused.

Bug, being the target addressed by a generated patch, is intrinsically tied to the patch's correctness. Numerous studies have delved into bug-focused information to discern the correctness of associated patches [2, 135, 136, 137, 124, 138, 139, 140, 141]. Compared with the aforementioned approaches, most of these bug-focused approaches tend to be dynamic as they execute passing or failing test suites to observe the behavior of the program. Xiong *et al.* [2] proposed to monitor the change in the execution traces of tests before and after the application of a patch. Their core intuition was that a correct patch would lead to substantial behavioral differences in failing test cases, while the behavior of passing test cases retain similar to the previous state. Acting on this presumption, they devised PATCH-SIM - a dynamic, test-based approach to identify overfitting patches by calculating the distance between the test executions on the original program and the patched program, eliminating the necessity for oracles. Martinez *et al.* [137] proposed a test-based clustering approach xTestCluster, which groups together patches that yield identical outputs when applied to the same failing test cases. The underlying argument is that patches demonstrating identical behavior for a given bug are semantically equivalent. As such, only one patch from each cluster needs to be reviewed by developers, optimising the patch verification process and saving valuable time and resources.

High-quality test suites are usually unavailable in practice, and the generated patches are prone to overfit to weak test suites. Yan *et al.* [136] proposed an approach that does not depend on user-provided test inputs. Instead, their method utilizes micro-execution to generate five distinct types of data: micro-trace code sequences, micro-trace value sequences, instruction position sequences, opcode/operand position sequences, and architecture sequences. This collected data serves as an approximation of the program's behavior, enabling the calculation of functional semantic similarity at the binary level. Finally, by embedding the gathered micro-execution features of the patches, they developed classifiers trained to determine the correctness of the patches. ROSE [138] retorts to the execution of an error-related routine on the current call stack and everything it calls, which are considered efficient for duplicating the buggy problem. Lee *et al.* [139] aim at validating Java null pointer exceptions (NPEs) patches when no tests are provided. Their approach first involves learning a null-handling model to predict and generalize the patterns developers use to fix issues. Using this predictive model, they proceed to obtain results from symbolic execution, which interprets NPE-triggering expressions in the buggy program, represented as the repair specification. Finally, the patched program's behavior is validated against the inferred specifications.

Despite notable advancements, existing techniques for patch correctness assessment have significant limitations. Dynamic methods, which often rely on test augmentation or runtime analysis, face two main issues. First, they struggle with the "test oracle problem," as accurate specifications for the expected output are frequently unavailable. Second, runtime analysis can be resource-intensive, requiring multiple executions of test cases. Consequently, these dynamic approaches are often

impractical for real-world use. On the static analysis front, whether pattern-based or learning-based, manual effort is indispensable for tasks like feature extraction, limiting the approaches' generalizability. Furthermore, two overarching challenges must be addressed to effectively assess patch correctness. The first challenge is the efficient representation of the patch's behavior. Existing methods based on syntax or semantics offer limited insights into the behavior of the code change. The second challenge revolves around the relationship between the bug and the patch. Current state-of-the-art techniques usually focus on the code changes and occasionally the test cases, but rarely examine the correspondence between the patch's behavior and the original bug. Our research seeks to assess patch correctness by specifically targeting the semantic behavior of correct patches in relation to the associated bug, thereby overcoming these two challenges.

3 Learning Representation of Code Changes for Patch Correctness

In this chapter, we study the benefit of learning code representations in order to learn deep features that may encode the properties of patch correctness. Our empirical work mainly investigates different representation learning approaches for code changes to derive embeddings that are amenable to similarity computations. We report on findings based on embeddings produced by pre-trained and re-trained neural networks. Experimental results demonstrate the potential of embeddings to empower learning algorithms in reasoning about patch correctness.

This chapter is based on the work published in the following research paper:

- **Haoye Tian**, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 981-992. 2020.

Contents

3.1 Overview

Automation in software engineering has recently reached new heights with the promising results recorded in the research direction of automated program repair (APR) [142, 9]. While a few techniques try to model program semantics and synthesize execution constraints towards producing quality patches, they often fail to scale to large programs. Instead, the large majority of research contributions [143] explore search-based approaches where patch candidates are generated and then validated against an oracle.

In the absence of strong program specifications, test suites represent affordable approximations that are widely used as the oracle in APR. In their seminal approach to test-based APR, Weimer *et al.* [144] considered that a patch is acceptable as soon as it makes the program pass all test cases in the test suite. Since then, a number of studies [25, 26] have explored the *overfitting* problem in patch validation: a given patch is synthesized to pass a test suite and yet is incorrect with respect to the intended program specification. Since limited test suites only weakly approximate program specifications, a patched program can indeed satisfy the requirements encoded in the test cases, and present a behavior outside of those tests that are significantly different from the behavior initially expected by the developer.

Overfitting patches constitute a key challenge in generate-and-validate APR approaches. Recent evaluation campaigns [145, 50, 132, 146, 46, 147, 107, 100, 19, 33] on APR systems are stressing on the importance of estimating the correctness ratio among the valid patches that can be found. To improve this ratio, researchers are investigating several research directions. We categorize them in three main axes that focus on actions before, during or after patch generation:

- *test-suite augmentation:* Yang *et al.* [34] proposed to generate better test cases to enhance the validation of patches, while Xin and Reiss [35] opted for increasing test inputs.
- *curation of repair operators:* approaches such as CapGen [46] successfully demonstrated that carefully-designed (e.g., fine-grained fix ingredients) repair operators can lead to more correct patches.
- *post-processing of generated patches:* Long and Rinard [38] studied some heuristics to discard patches that are likely overfitting.

Our work is related to the latter thrust. So far, the state-of-the-art works targeting the identification of patch correctness are mainly implemented based on computing the similarity of test case execution traces [2]. Ye *et al.* [148] followed up by presenting preliminary results suggesting that statically-extracted code features at the syntax level could be used to predict overfitting patches. While such an approach is appealing, the feature engineering effort can be huge when researchers target generalizable approaches. To cope with this problem, Csuvik *et al.* [125] have proposed a preliminary small-scale study on the use of embeddings: leveraging pre-trained natural language sentence embedding models, they claim to have been able to filter out 45% incorrect patches generated for 40 bugs from the QuixBugs dataset [149].

This paper. Embeddings have been successfully applied to various prediction tasks in software engineering research [150, 151, 13, 152]. For patch correctness prediction, the literature does not yet provide extensive experimental results to guide future research. Our work fills this gap. We investigate in this paper the feasibility

of leveraging advances in deep representation learning to produce embeddings that are amenable to reasoning about correctness.

❶ We investigate different representation learning models adapted to natural language tokens and source code tokens that are more specialized to code changes. Our study considers both pre-trained models and the retraining of models.

❷ We empirically investigate whether, with learned representations, the hypothesis of minimal changes incurred by correct patches remains valid: experiments are performed to check the statistical difference between similarity scores yielded by correct patches and those yielded by incorrect patches.

❸ We run exploratory experiments assessing the possibility to select cutoff similarity scores between learned representations of buggy code and patched code fragments for heuristically filtering out incorrect patches.

❹ Finally, we investigate the discriminative power of deep learned features in a classification training pipeline aimed at learning to predict patch correctness.

3.2 Background

Our work deals with various concepts and techniques from the fields of program repair and machine learning. We present the relevant details in this section to facilitate readers' understanding of our study design and the scope of our experiments.

3.2.1 Patch Plausibility and Correctness

Defining patch correctness is a non-trivial challenge in automated program repair. Until the release of empirical investigations by Smith *et al.* [26], actual correctness (w.r.t. program behavior intended by developers) was seldom used as a performance criterion of patch generation systems. Instead, experimental results were focused on the number of patches that make the program pass all test cases. Such patches are actually only **plausible**. Qi *et al.* [25] demonstrated in their study that an overwhelming majority of plausible patches generated by GenProg [153], RSRepair [52] and AE [75]) are overfitting the test suite while actually being incorrect. To improve the probability of program repair systems to generate **correct** patches, researchers have mainly invested in strengthening the validation oracle (i.e., the test suites). Opad [34], DiffTGen [35], UnsatGuided [1], PATCH-SIM/TEST-SIM [2] generate new test inputs that trigger behavior cases which are not addressed by APR-generated patches.

More recent works [148, 125] are starting to investigate static features and heuristics (or machine learning) to build predictive models of patch correctness. Ye *et al.* [148] presented the ODS approach which relates to our study since it investigated machine learning with static features extracted from Java program patches. Their approach however builds on carefully hand-crafted features, which may not generalize to other programming languages or even to varied datasets. The study of Csuvik *et al.* [125] is also closely related to ours since it explores BERT embeddings to define similarity thresholds. Their work however remains preliminary (it does not investigate the discriminative power of features) and has been performed at a very small scale (single pre-trained model on 40 one-line bugs from simple programs).

3.2.2 Distributed Representation Learning

Learning distributed representations have been widely used to advance several machine learning-based software engineering tasks [154, 155, 156, 157, 158]. In particular, embedding techniques such as **Word2Vec** [159], **Doc2Vec** [159] and **BERT** [155] have been successfully applied to different semantics-related tasks such as code clone detection [160], vulnerability detection [161], code recommendation [162], and commit message generation [163].

By building on the hypothesis of code naturalness [164, 165], a number of software engineering research works have also leveraged the aforementioned approaches for learning distributed representations of code [166, 13]. Alon *et al.* [167] have then proposed **code2vec**, an embedding technique that explores AST paths to take into account structural information in code. More recently, Hoang *et al.* [163] have proposed **CC2Vec**, which further specializes to code changes. Our work explores different techniques across the spectrum of distributed representation learning. We therefore consider four variants from the seemingly-least specialized to code (i.e., Doc2Vec) to the state of the art for code change representation (i.e., CC2Vec).

Doc2Vec [159] is an unsupervised framework mostly used to learn continuous

distributed vector representations of sentences, paragraphs and documents, regardless of their lengths. It works on the intuition, inspired by the method of learning word vectors [168], that the document representation should be good enough to predict the words in the document. Doc2Vec has been applied in various software engineering tasks. For example, Wei and Li [160] leveraged Doc2Vec to exploit deep lexical and syntactical features for software functional clone detection. Ndichu *et al.* [161] employed Doc2Vec to learn code structure representation at AST level to predict JavaScript-based attacks.

BERT [155] is a language representation model that has been introduced by an AI language team in Google. BERT is devoted to pre-train deep bidirectional representations from unlabelled texts. Then a pre-trained BERT model can be fine-tuned to accomplish various natural language processing tasks such as question answering or language inference. Zhou *et al.* [162] employed a BERT pre-trained model to extract deep semantic features from code name information of programs in order to perform code recommendation. Yu *et al.* [169] even leveraged BERT on binary code to identify similar binaries.

code2vec [167] is an attention-based neural code embedding model developed to represent code fragments as continuous distributed vectors, by training on AST paths and code tokens. Its embeddings have notably been used to predict the semantic properties of code fragments [167], in order, for instance, to predict method names. Compton *et al.* [170] recently leveraged code2vec to embed Java classes and learn code structures for the task of variable naming obfuscation.

CC2Vec [163] is a specialized hierarchical attention neural network model which learns vector representations of code changes (i.e., patches) guided by the associated commit messages (which is used as a semantic representation of the patch). As the authors demonstrated in their large empirical evaluation, CC2Vec presents promising performance on commit message generation, bug fixing patch identification, and just-in-time defect prediction.

3.3 Study Design

First, we overview the research questions that we investigate. Then we present the datasets that are leveraged to answer these research questions. Finally, we discuss the actual training of (or use of pre-trained) models for embedding the code changes.

3.3.1 Research Questions

RQ1: *Do different representation learning models yield comparable distributions of similarity values between buggy code and patched code?* A widespread hypothesis in program repair is that bug fixing generally induce minimal changes [43, 44, 45, 132, 145, 50, 46, 144, 171, 18, 42]. We propose to investigate whether embeddings can be a reliable means for assessing the extent of changes through computation of cosine similarity between vector representations.

RQ2: *To what extent similarity distributions can be generalized for inferring a cutoff value to filter out incorrect patches?* Following up on RQ1, We propose in this research question to experiment ranking patches based on cosine similarity of their vector representations, and rely on naively-defined similarity thresholds to decide on filtering of incorrect patches.

RQ3: *Can we learn to predict patch correctness by training classifiers with code embeddings input?* We investigate whether deep learned features are indeed relevant for building machine learning predictors for patch correctness.

3.3.2 Datasets

We collect patch datasets by building on previous efforts in the community. An initial dataset of correct patches is collected by using five literature benchmarks, namely Bugs.jar [172], Bears [173], Defects4J [88], QuixBugs [92] and ManySStuBs4J [174]. These are developer patches as committed in open source project repositories.

We also consider patches generated by APR tools integrated into the RepairThemAll framework. We use all patch samples released by Durieux *et al.* [28]. This only includes sample patches that make the programs pass all test cases. They are thus plausible. However, no validation information on correctness was given. In this work, we proceed to manually validate the generated patches, among which we identified 900 correct patches. The correctness validation follows the criteria defined by Liu *et al.* [175], which involves comparing the APR-generated patch to the developer-provided patch found in the benchmark.

In a recent study on the efficiency of program repair, Liu *et al.* [175] released a labeled dataset of patches generated by 16 APR systems for the Defects4J bugs. We consider this dataset as well as the labeled dataset that was used to evaluate the PATCH-SIM [2] approach.

Overall, Table 3.1 summarizes the data sets that we used for our experiments. Each experiment in Section 3.4 has specific requirements on the data (e.g., large patch sets for training models, labeled datasets for benchmarking classifiers, etc.). For each experiment, we will recall which sub-dataset has been leveraged and why.

3.3.3 Model input pre-processing

Samples in our datasets are patches such as the one presented in Figure 3.1 extracted from the Defects4J dataset. Our investigations with representation learning however require input data about the buggy and patched code. A straightforward

Table 3.1: Datasets of Java patches used in our experiments.

Subjects	contains incorrect patches	contains correct patches	labelled dataset	# Patches
Bears [173]	No	Yes	-	251
Bugs.jar [172]	No	Yes	-	1,158
Defects4J [88]†	No	Yes	-	864
ManySStubBs4J [174]	No	Yes	-	34,051
QuixBugs [92]	No	Yes	-	40
RepairThemAll [28]	Yes	Yes	No‡	64,293
Liu *et al.* [175]	Yes	Yes	Yes	1,245
Xiong *et al.* [2]	Yes	Yes	Yes	139
Total				102,041

†The latest version 2.0.0 of Defects4J is considered in this study.
‡The patches are not labeled in [28]. We support the labeling effort in this study by comparing the generated patches against the developers' patches. The 2,918 patches for IntroClassJava in [28] are also excluded from our study since IntroClassJava is a lab-built Java benchmark transformed from the C program bugs in small student-written programming assignments from IntroClass [83].

approach to derive those inputs would be to consider the code files before and after the patch. Unfortunately, depending on the size of the code file, the differences could be too minimal to be captured by any similarity measurement. To that end, we propose to focus on the code fragment that appears in the patch. Thus, to represent the buggy code fragment (cf. Figure 3.2), we keep all removed lines (i.e., starting with '-') as well as the patch context lines (i.e., those not starting with either '-', '+' or '@'). Similarly, the patched code fragment (cf. Figure 3.3) is represented by added lines (i.e., starting with '+') as well as the same context lines. Since tool support for the representation learning techniques BERT, Doc2Vec, and CC2Vec require each input sample to be on a single line, we flatten multi-line code fragments into a single line.

```
--- source/org/jfree/chart/renderer/category/
    AbstractCategoryItemRenderer.java
+++ source/org/jfree/chart/renderer/category/
    AbstractCategoryItemRenderer.java
@@ -1795,6 +1795,6 @@ public abstract class
    AbstractCategoryItemRenderer
          int index = this.plot.getIndexOf(this);
          CategoryDataset dataset = this.plot.getDataset(index);
-         if (dataset != null) {
+         if (dataset == null) {
              return result;
          }
```

Figure 3.1: Example of a patch for the Defects4J bug Chart-1.

In contrast to BERT, Doc2Vec, and CC2Vec, which can take as input some syntax-incomplete code fragments, code2vec requires the fragment to be fully parsable in order to extract information on Abstract Syntax Tree paths. Since patch datasets include only text-based diffs, code context is generally truncated and is likely not parsable. However, as just explained, we opt to consider only the removed/added lines to build the buggy and patched code input data. By doing so, we substantially improved the success rate of the JavaExtractor tool used to build the tokens in the code2vec pipeline.

3.3.4 Embedding models

When representation learning algorithms are applied to some training data, they produce *embedding models* that have learned to map a set of code tokens in the vocabulary of the training data to vectors of numerical values. These vectors are

```
1: a/source/org/jfree/chart/renderer/category/
   AbstractCategoryItemRenderer.java
2:         int index = this.plot.getIndexOf(this);
3:         CategoryDataset dataset = this.plot.getDataset(index)
   ;
4:         if (dataset != null) {
5:             return result;
6:         }
```

Figure 3.2: Buggy code fragment associated to patch in Fig. 3.1.

```
1: b/source/org/jfree/chart/renderer/category/
   AbstractCategoryItemRenderer.java
2:         int index = this.plot.getIndexOf(this);
3:         CategoryDataset dataset = this.plot.getDataset(index)
   ;
4:         if (dataset == null) {
5:             return result;
6:         }
```

Figure 3.3: Patched code fragment associated to patch in Fig. 3.1.

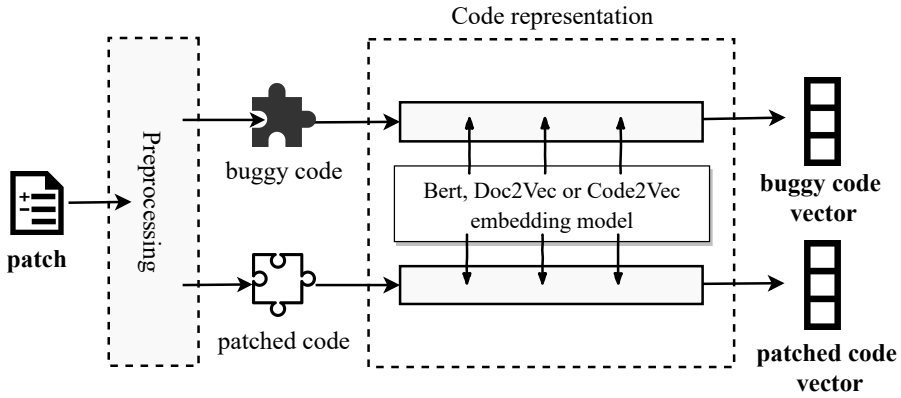


Figure 3.4: Producing code fragment learned embeddings with BERT, Doc2Vec and code2vec.

also referred to as *embeddings*. Figure 3.4 illustrates the process of embedding buggy code and patched code for the purpose of our experiments.

The embedding models used in this work are obtained from different sources and training scenarios.

- **BERT.** In the first scenario, we consider an embedding model that initially targets natural language data, both in terms of the learning algorithm and in terms of training data. The network structure of BERT, however, is deep, meaning that it requires large datasets for training the embedding model. As it is now custom in the literature, we instead leverage a pre-trained 24-layer BERT model, which was trained on a Wikipedia corpus.
- **Doc2Vec.** In the second scenario, we consider an embedding model that is trained on code data but using a representation learning technique that was developed for text data. To that end, we have trained the Doc2Vec model with code data of 36,364 patches from the 5 repair benchmarks (cf. Table 3.1).
- **code2vec.** In the third scenario, we consider an embedding model that primarily targets code, both in terms of the learning algorithm and in terms of training data. We use in this case a pre-trained model of code2vec, which was trained by the authors using ~14 million code examples from Java projects.

- **CC2Vec.** Finally, in the fourth scenario, we consider an embedding model that was built in representation learning experiments for code changes. However, the pre-trained model that we leveraged from the work of Hoang *et al.* [163] is embedding each patch into a single vector. We investigate the layers and identify the middle CNN-3D layer as the sweet spot to extract embeddings for buggy code and patched code fragments. Figure 3.5 illustrates the process.

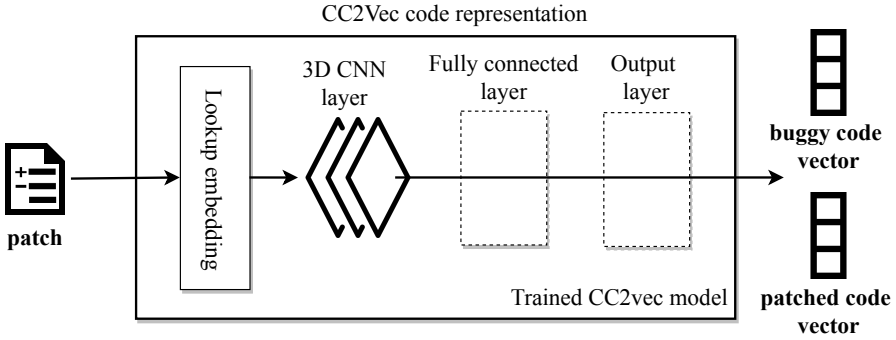


Figure 3.5: Extracting code fragment learned embeddings from CC2Vec pre-trained model.

3.4 Experiments and Results

We present the experiments that we designed to answer the research questions of our study. For each experiment, we state the objective, overview the execution details before presenting the results.

3.4.1 RQ-1: Similarity Measurements for Buggy and Patched Code using Embeddings

[Objective]: We investigate the capability of different learned embeddings to capture the (dis)similarity between buggy code fragments and the (in)correctly-patched ones. The experiments are performed towards providing answers for two sub-questions:

- RQ-1.1 *Is correctly-patched code actually similar to buggy code based on learned embeddings?*
- RQ-1.2 *To what extent is buggy code more similar to correctly-patched code than to incorrectly-patched code?*

[Experimental Design for RQ-1.1]: Using the four embedding models considered in our study (cf. Section 3.3.4), we produce the learned embeddings for buggy and patched code fragments associated to 36k patches from five repair benchmarks shown in Table 3.2. In this case, the patched code fragment is the correctly-patched code fragment since it comes from labeled benchmark data (generally representing human-written patches). Given those learned embeddings (i.e., deep learned representation vectors of code), we compute the cosine similarity between the vectors representing the buggy and correctly-patched code fragments.

Table 3.2: Datasets used for assessing the similarity between buggy code and correctly-patched code.

	Bears	Bugs.jar	Defects4J	ManySStuBs4J	QuixBugs	Total
# Patches	251	1,158	864	34,051	40	36,364[1]

[Experimental Results for RQ-1.1]: Figure 3.6 presents the boxplots of the similarity distributions with different embedding models and for samples in different datasets. Doc2Vec and code2vec models appear to yield similarity values that are lower than BERT and CC2Vec models.

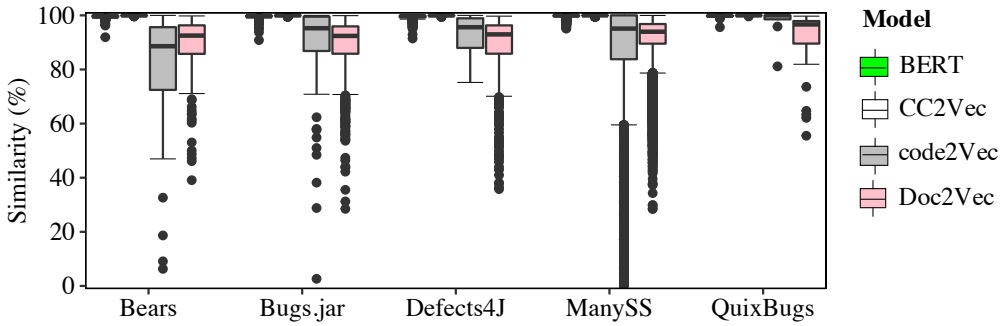
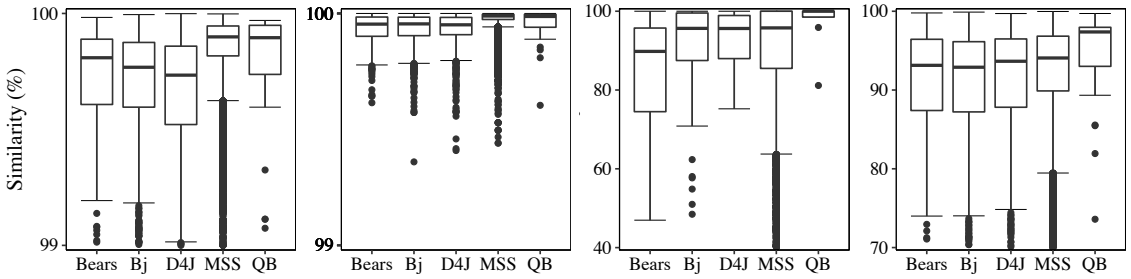


Figure 3.6: Distributions of similarity scores between correctly-patched code fragments and buggy ones.

Figure 3.7 zooms in the boxplot region for each embedding model experiment to overview the differences across different benchmark data. We obverse that, when embedding the patches with BERT, the similarity distribution for the patches in

Defects4J dataset is similar to Bugs.jar and Bears dataset, but is different from the dataset ManySStBs4J and QuixBugs. The Mann-Whitney-Wilcoxon (MWW) tests [176, 177] confirm that the similarity of median scores for Defects4J, Bugs.jar and Bears is indeed statistically significant. MWW tests further confirms the statistical significance of the difference between Defects4J and ManySStBs4J/QuixBugs scores.



(a) BERT. (b) CC2Vec. (c) code2vec. (d) Doc2Vec.

Figure 3.7: Zoomed views of the distributions of similarity scores between correct and buggy code fragments.

Defects4J, Bugs.jar and Bears include diverse human-written patches for a large spectrum of bugs from real-world open-source Java projects. In contrast, ManySStuBs4J only contains patches for single statement bugs. Quixbugs dataset is further limited by its size and the fact that the patches are built by simply mutating the code of small Java implementation of 40 algorithms (e.g., quicksort, levenshtein, etc.).

While CC2Vec and Doc2Vec exhibit roughly similar performance patterns with BERT (although at different scales), the experimental results with code2vec present different patterns across datasets. Note that, due to parsing failures of code2vec, we eventually considered only 118 Bears patches, 123 Bugs.jar patches, 46 Defects4J patches, 20,840 ManySStuBs4J patches and 8 QuixBugs. The change of dataset size could explain the difference with the other embedding models.

✍ **RQ-1.1** ▶ *learned embeddings of buggy and correctly-patched code fragments exhibit high cosine similarity scores. Median scores are similar for patches that are collected with similar heuristics (e.g., in-the-wild patches vs single-line patches vs debugging example patches). The pre-trained BERT natural language model captures more similarity variations than the CC2Vec model, which is specialized for code changes.*◀

[**Experimental Design for RQ-1.2**]**:** To compare the similarity scores of correctly-patched code fragment vs incorrectly-patched code fragment to the buggy one, we consider combining datasets with correct patches and datasets with incorrect patches. Note that, all patches in our experiments are plausible since we are focused on correctness: plausibility is straightforward to decide based on test suites. Correct patches are provided in benchmarks. However, all the benchmarks in our study do not contain incorrect patches. Therefore, we rely on the dataset released by Liu *et al.* [175]: 674 plausible but incorrect patches generated by 16 repair tools for 184 Defects4J bugs are considered from this dataset. Those 674 incorrect patches are selected within a larger set of incorrect patches by adding the constraint that the incorrect patch should be changed the same code location as the developer-

[1]Due to parsing failures, code2vec learned embeddings are available for 21,135 patches.

Table 3.3: Scenarios for similarity distributions comparison.

Scenario	Incorrect patches	Correct patches
Imbalanced-all[2]	674 incorrect patches	36,364 correct patches from 5 benchmarks in Table 3.2.
Imbalanced-Defects4J	by 16 APR tools [175]	864 correct patches from Defects4J.
Balanced-Defects4J	for 184 Defects4J bugs.	184 correct patches for the 184 Defects4J bugs.

provided patch in the benchmark: such incorrect patch cases may indeed be the most challenging to identify with heuristics.

We consider three scenarios to select correct patches for the comparison of the similarity scores. (1) Imbalanced-all, a quick intuition is that we compare the 674 incorrect patches against all correct patches from 5 benchmarks. (2) Imbalanced-Defects4J, we only use the correct patches from Defects4J. We design the second scenario because the correct patches from other benchmarks may create a sample bias. (3) Balanced-Defects4J, we use the correct patches for the 184 Defects4J bugs that the 674 incorrect patches target. In this scenario, incorrect and correct sets have the same number of patches. We design this to avoid the underlying bias of imbalanced sets. The comparison is done with different scenarios specified in Table 3.3.

[Experimental Results for RQ-1.2]: In this experiment, we further assess whether incorrectly-patched code exhibits different similarity score distributions than correctly-patched code. Figure 3.8 shows the distributions of cosine similarity scores for correct patches (i.e., similarity between buggy code fragments and correctly-patched ones) and incorrect patches (i.e., similarity between buggy code fragments and incorrectly-patched ones). The comparison is done with different scenarios specified in Table 3.3.

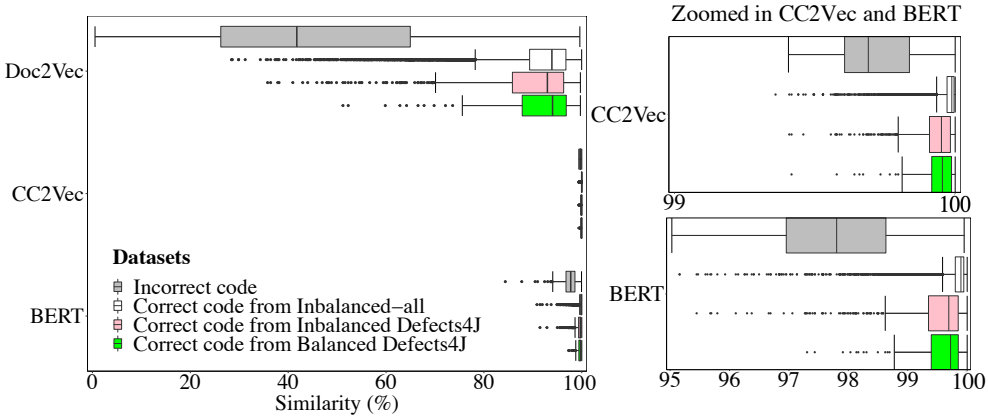


Figure 3.8: Comparison of similarity score distributions for code fragments in incorrect and correct patches.

The comparisons do not include the case of learned embeddings for code2vec. Indeed, unlike the previous experiment where code2vec was able to parse enough code fragments, for the considered 184 correct patches of Defects4J, code2vec failed to parse most of the relevant code fragments. Hence, we focus the comparison on the other three embedding models (pre-trained BERT, trained Doc2Vec and pre-trained CC2Vec). Overall, we observe that the distribution of cosine similarity scores is substantially different for correctly-patched and incorrectly-patched code fragments.

We observe that the similarity distributions of buggy code and patched code from incorrect patches are significantly different from the similarities for correct patches. The difference of median values is confirmed to be statistically significant by an

[2]Except for Defects4J, there are no publicly-released incorrect patches for APR datasets.

MWW test. Note that the difference remains high for BERT, Doc2Vec and CC2Vec whether the correctly-patched code is the counterpart of the incorrectly-patched ones (i.e., the scenario of Balanced-Defects4J) or whether the correctly-patched code is from a larger dataset (i.e., Imbalanced-Defects4J scenarios). As for the comparison with the dataset of Imbalanced-all, the heuristic remains valid but note it may be affected by other benchmarks, i.e., the different bugs caused the results.

✍ **RQ-1.2** ► *learned embeddings of code fragments with BERT, CC2Vec and Doc2Vec yield similarity scores that, given a buggy code, substantially differ between correctly-patched code and incorrectly-patched one. This result suggests that similarity score can be leveraged to discriminate correct patches from incorrect patches.*◄

3.4.2 RQ-2: Filtering of Incorrect Patches based on Similarity Thresholds

[Objective]: Following up on the findings related to the first research question, we investigate the selection of cut-off similarity scores to decide on which APR-generated patches are likely incorrect. Results from this investigation will provide insights to guide the exploitation of code learned embeddings in program repair pipelines.

[Experimental Design]: To select threshold values, we consider the distributions of similarity scores from the above experiments (cf. Section 3.4.1). Table 3.4 summarizes relevant statistics on the distributions on the similarity scores distribution for correct patches. Given the differences that were exhibited with incorrect patches in previous experiments, we use, for example, the 1^{st} quartile value as an inferred threshold value.

Table 3.4: Statistics on the distributions of similarity scores for correct patches of Bears+Bugs.jar+Defects4J.

Subjects	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean
BERT	90.84	99.47	99.73	99.86	100	99.54
CC2Vec	99.36	99.91	99.95	99.98	100	99.93
Doc2Vec	28.49	85.80	92.60	96.10	99.89	89.19
code2vec	2.64	81.19	93.63	98.87	100	87.11

Given our previous findings that different datasets exhibit different similarity score distributions, we also consider inferring a specific threshold for the QuixBugs dataset (cf. statistics in Table 3.5).

Table 3.5: Statistics on the distributions of similarity scores for correct patches of QuixBugs.

Subjects	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean
BERT	95.63	99.69	99.89	99.95	99.97	99.66
CC2Vec	99.60	99.94	99.99	100	100	99.95
Doc2Vec	55.51	89.56	96.65	97.90	99.72	91.29
code2vec	81.16	98. 53	100	100	100	97.06

Our test data is constituted of 64,293 patches generated by 11 APR tools in the empirical study of Durieux *et al.* [28]. First, we use the four embedding models to generate learned embeddings of buggy code and patched code fragments and compute cosine similarity scores. Second, for each bug, we rank all generated patches

Table 3.6: Filtering incorrect patches by generalizing thresholds inferred from Section 3.4.1.Results.

Dataset		Bears,Bugsjar,Defects4J		QuixBugs	
# Correct Patches		893		7	
# Incorrect Patches		61,932		1,461	
Model/Metric/Threshold		1st Qu.	Mean	1st Qu.	Mean
BERT	# +CP	57	49	4	4
	# -IP	48,846	51,783	1,387	1,378
	+Recall	6.4%	5.5%	57.1%	57.1%
	-Recall	78.9%	83.6%	94.9%	94.3%
CC2Vec	# +CP	797	789	4	4
	# -IP	19,499	23,738	1,198	1,255
	+Recall	89.2%	88.4%	57.1%	57.1%
	-Recall	31.5%	38.3%	82.0%	85.9%
Doc2Vec	# +CP	794	771	7	7
	# -IP	25,192	33,218	1,226	1,270
	+Recall	88.9%	86.3%	100%	100%
	-Recall	40.7%	53.6%	83.9%	86.9%

"**# +CP**" means the number of correct patches that can be ranked upon the threshold, while "**# -IP**" means the number of incorrect patches that can be filtered out by the threshold. "**+Recall**" and "**-Recall**" represent the recall of identifying correct patches and filtering out incorrect patches, respectively.

based on the similarity scores between the patched code and the buggy one, where we consider that the higher the score, the more likely the correctness. Finally, to filter incorrect candidates, we consider two experiments:

1. Patches that lead to similarity scores that are lower to the inferred threshold (i.e., 1^{st} quartile in previous experimental data) will be considered as incorrect. Patches where patched code exhibit higher similarity scores than the threshold are considered correct.
2. Another approach is to consider only the top-1 patches with the highest similarity scores as correct patches. Other patches are considered incorrect.

In all cases, we systematically validate the correctness of all 64,293 patches to have the correctness labels, for which the dataset authors did not provide (all plausible patches having been considered as valid). First, if the file(s) modified by a patch are not the same buggy files in the benchmark, we systematically consider it as incorrect: with this simple scheme, 33,489 patches are found incorrect. Second, with the same file, if the patch is not making changes at the same code locations, we consider it to be incorrect: 26,386 patches are further tagged as incorrect with this decision (cf. Threats to validity in Section 3.5). Finally, for the remaining 4,418 plausible patches in the dataset, we manually validate correctness by following the strict criteria enumerated by Liu *et al.* [175] to enable reproducibility. Overall, we could label 900 correct patches. The remainders are considered as incorrect.

[Experimental Results]: By considering the patch with the highest (top-1) similarity score between the patched code and buggy code as correct, we were able to identify a correct patch for 10% (with BERT), 9% (with CC2Vec) and 10% (with Doc2Vec) of the bug cases. Overall we also misclassified 96% correct patches as incorrect. However, only 1.5% of incorrect patches were misclassified as correct patches.

Given that a given bug can be fixed with several correct patches, the top-1 criterion may not be adequate. Furthermore, this criterion makes the assumption

that a correct patch indeed exists among the patch candidates. By using filtering thresholds inferred from previous experiments (which do not include the test dataset in this experiment), we can attempt to filter all incorrect patches generated by APR tools. Filtering results presented in Table 3.6 show the recall scores that can be reached. We provide experimental results when we use 1^{st} quartile and Mean values of similarity scores in the "training" set as threshold values. The thresholds are also applied by taking into account the datasets: thresholds learned on QuixBugs benchmark are applied to generated patches for QuixBugs bugs.

✍ **RQ-2** ►*Building on cosine similarity scores, code fragment learned embeddings can help to filter out between 31.5% with CC2Vec and 94.9% with BERT of incorrect patches. While BERT achieves the highest recall of filtering incorrect patches, it produces learned embeddings that lead to a lower recall (at 5.5%) at identifying correct patches.*◄

3.4.3 RQ-3: Classification of Correct Patches with supervised learning

[Objective]: Cosine similarity between embeddings (which was used in the previous experiments) considers every deep learned feature as having the same weight as the others in the embedding vector. We investigate the feasibility to infer, using machine learning, the weights that different features may present with respect to patch correctness. We compare the prediction evaluation results with the achievements of related approaches in the literature.

[Experimental design]: To perform our machine learning experiments, we first require a ground-truth dataset. To that end, we rely on labeled datasets in the literature. Since incorrect patches generated by actual APR tools are only available for the Defects4J bugs, we focus on labeled patches provided by two independent teams (Liu *et al.* [175] and Xiong *et al.* [2]). Very few patches generated by the different tools are actually labeled as correct, leading to an imbalanced dataset. To reduce the imbalance issue, we supplement the dataset with developer (correct) patches as supplied in the Defects4J benchmark. Eventually, our dataset shown in Table 3.7 included 1000 patches after removing duplicates to avoid data bias.

Table 3.7: Dataset for evaluating ML-based predictors of patch correctness.

	Correct patches	**Incorrect patches**	**Total**
Liu *et al.* [175]	137	502	639
Xiong *et al.* [2]	30	109	139
Defects4J (developers) [88]	356	0	356
Whole dataset	523	611	1134
Final Dataset (deduplicated)	**468**	**532**	**1000**

Our ground truth dataset patches are then fed to our embedding models to produce embedding vectors. As for previous experiments, the parsability of Defects4J patch code fragments prevented the application of code2vec: we use pre-trained models of BERT (trained with natural language text) and CC2Vec (trained with code changes) as well as a retrained model of Doc2Vec (trained with patches).

Since the representation learning models are applied to code fragments inferred from patches (and not to the patch themselves), we collect the embeddings of both buggy code fragment and patched code fragment for each patch. Then we must merge these vectors back into a single input vector for the classification algorithm.

We follow an approach that was demonstrated by Hoang et al. [163] in a recent work on bug fix patch prediction: the classification model performs best when features of patched code fragment and buggy code fragment are crossed together. We thus propose a classification pipeline (cf. Figure 3.9) where the feature extraction for a given patch is done by applying subtraction, multiplication, cosine similarity and euclidean similarity to capture crossed features between the buggy code vector and the patched code vector. The resulting patch embedding has 2*n+2 dimensions where n is the dimension of input code fragment embeddings. The values of the dimension n for BERT, Doc2Vec and CC2Vec are set as 1024, 64 and 64, respectively.

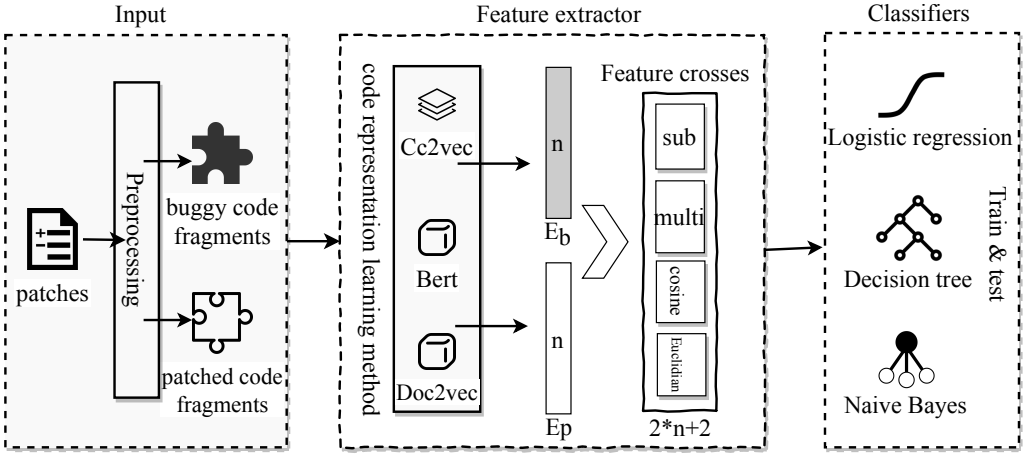


Figure 3.9: Feature engineering for correctness classification.

[**Experimental Results**]**:** We compare the performance of different predictors (varying the embeding models) using different learners (i.e., classification algorithms). Results presented in Table 3.8 are averaged from a 5-fold cross validation setup. All classical metrics used for assessing predictors are reproted: Accuracy, Precision, Recall, F1-Measure, Area Under Curve (AUC). Logistic Regression (LR) applied to BERT embeddings yield the best performance measurements: 0.720 for F1 and 0.808 for AUC.

Table 3.8: Evaluation of representation models on three ML classifiers.

Classifier	Embedding	Acc.	Prec.	Recall.	F1	AUC
DecisionTree	BERT	63.6	62.0	57.3	59.6	0.632
	CC2Vec	69.0	66.9	68.0	67.2	0.690
	Doc2Vec	60.2	57.4	57.7	57.5	0.600
Logistic regression	BERT	74.4	73.8	70.3	72.0	0.808
	CC2Vec	73.9	72.5	72.0	72.0	0.788
	Doc2Vec	66.3	65.3	59.9	62.3	0.707
Naive bayes	BERT	60.3	55.6	77.0	64.5	0.642
	CC2Vec	58.0	65.4	22.7	28.5	0.722
	Doc2Vec	66.3	69.4	49.8	57.9	0.714

✍ **RQ3.1** ▶ *An ML classifier trained using Logistic Regression with BERT embeddings yield very promising performance on patch correctness prediction (F-Measure at 72.0% and AUC at 80.8%).* ◀

[Comparison against the state of the art]. There are two related works for patch prediction which were both evaluated on 139 patches released by

Table 3.9: Comparison of incorrect patch identification between PATCH-SIM (uses dynamic information) and BERT+ LR (uses embeddings statically inferred from patches).

	Ground Truth		PATCH-SIM		BERT + LR	
Project	**Incorrect**	**Correct**	**Incorrect excluded (%)**	**Correct excluded**	**Incorrect excluded (%)**	**Correct excluded**
Chart	23	3	14(60.9%)	0	16(69.6%)	0
Lang	10	5	6(54.5%)	0	1(10%)	0
Math	63	20	33(52.4%)	0	23(36.5%)	0
Time	13	2	9(69.2%)	0	3(23.1%)	0
Total	109	30	62(56.3%)	0	43(39.4%)	0

Xiong *et al.* [2]. PATCH-SIM [2] compares execution traces of patched programs to identify correctness. ODS [148] leverages manually-crafted features to build machine learning classifiers.

We consider the 139 patches as test set and the remainder in our dataset ($870 = 1000 - 130$[3]) for training. Note that the 139 patches are associated to bug cases where repair tools can generate patches. These patches may thus be substantially different from the rest in our dataset. Indeed our best learner (Logistic Regression with BERT embeddings) yields an AUC of 0.765, i.e., the overall capability of classifying correct patches. The Receiver Operating Characteristic (ROC) curve is presented in Figure 3.10.

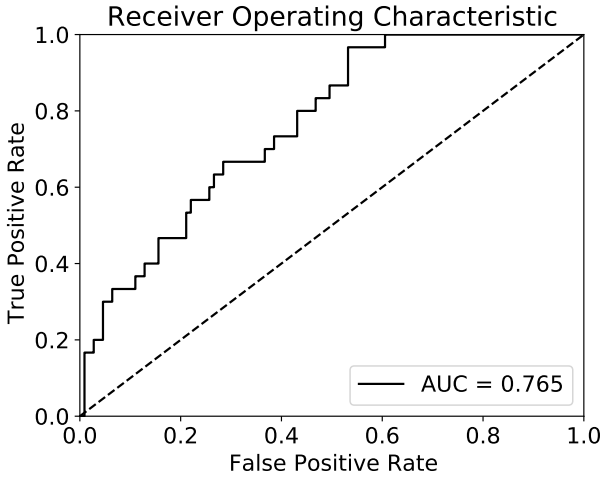


Figure 3.10: Performance of ML patch correctness predictor using BERT/Logistic Regression: Test set from [2].

In the validation of PATCH-SIM [2], the authors aimed for avoiding to filter out any correct patches. Eventually, when guaranteeing that no correct patch is excluded, they could still exclude 62 (56.3%) incorrect patches. If we constrain the threshold of our predictor to avoid misclassifying any correct patch (threshold value = 0.219), our predictor is able to exclude up to 43 (39.4%) incorrect patches, which represents a reasonably promising achievement since no dynamic information is used (in contrast to PATCH-SIM). Table 3.9 overviews the prediction results comparison.

We also compare the predictive power of our models against that of ODS [148], which builds on manually engineered features. We directly compare against the results reported by the authors on the 139 test patches. While the pre-trained BERT

[3]9 patches in the ground truth dataset by Xiong et al. [2] were duplicates (e.g., Patch151 ≡ Patch23).

Table 3.10: Confusion matrix of ML predictions based on BERT embedddings with different thresholds.

Learners	AUC		Thresholds								
			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
LR	0.765	#TP	30	30	24	19	16	12	10	6	4
		#TN	13	37	61	79	85	95	100	106	108
		#FP	96	72	48	30	24	14	9	3	1
		#FN	0	0	6	11	14	18	20	24	26
RF	0.751	#TP	30	30	29	26	20	12	4	2	0
		#TN	1	1	6	32	79	102	107	108	109
		#FP	108	108	103	77	30	7	2	1	0
		#FN	0	0	1	4	10	18	26	28	30

Table 3.11: Confusion matrix of ODS predictions with different thresholds.

Learners	AUC		Thresholds								
			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
LR	0.705	#TP	27	27	27	27	27	27	27	27	27
		#TN	50	50	50	50	50	51	51	52	52
		#FP	60	60	60	60	60	59	59	58	58
		#FN	2	2	2	2	2	2	2	2	2
RF	0.841	#TP	29	29	29	29	29	29	25	23	14
		#TN	20	33	36	43	51	60	68	81	101
		#FP	90	77	74	67	59	50	42	29	13
		#FN	0	0	0	0	0	0	4	6	15

model associated with Logistic Regression (LR) achieves better AUC than ODS LR-based model (0.765 vs 0.705), ODS Random Forest-based model achieves a higher AUC at 0.841. Note however that ODS has been trained on over 13 thousand patches (including patches for bugs associated to the test set patch), our training dataset includes only 870 patches (i.e., $\sim 1/20^{th}$ of their dataset).

Tables 3.10 and 3.11 provide confusion matrices for different cut-off thresholds of the classifiers for ODS and our BERT embeddings-based classifiers: TP (true positives) represent correct patches that were classified as such; TN (true negatives) represent incorrect patches that were classified as such; FP (false positives) represent incorrect patches that were classified as correct; and FN (false negatives) represent correct patches that were classified as incorrect. Overall, the BERT-based predictor is very sensitive to the cut-off thresholds while ODS is less sensitive. We also note that BERT embeddings applied to Random Forrest does not yield good performance: decision trees are indeed known to be good for categorical data and request large datasets for training. In our case, the data set is small, while ODS has a training dataset that is about 20 times larger. The hand-crafted features of ODS may also help split the patches into categories while our deep learned features are based on a large vocabulary of natural language text.

We observe nevertheless that LR classifiers fed with BERT embeddings are able to recall high numbers of incorrect patches (#TN is high and #FP is low on threshold > 0.5). In contrast ODS consistently recalls correct patches (however with high false positives). These experimental results suggest that both approaches can be used in a complementary way. In future work, we will propose an approach that carefully merges deep learned features to hand-crafted features towards yielded a better predictors of patch correctness.

✍ **RQ3.2** ►*ML predictors trained on learned representations appear to perform slightly less well than state of the art PATCH-SIM approach which relies on dynamic information. On the other hand, deep code representations appear to be complementary to hand-crafted features engineered for ODS. Overall, we recall that our experimental evaluations are performed in a zero-shot scenario, i.e., without fine-tuning the parameters of any of the pre-trained models. Furthermore, the training dataset of the classifiers is an order of magnitude smaller*[a] *than the one used by most closely-related work (i.e., ODS) and may further not be representative to best fit the test set.*◄

[a]We were not able to collect or reconstitute the training dataset used in ODS to train our model.

3.5 Discussions

We enumerate a few insights from our experiments with representation learning models and discuss some threats to validity.

3.5.1 Experimental Insights

Code-oriented embedding models may not yield the best embeddings for training patch correctness classifiers. Our experiments have revealed that the BERT model, which was pre-trained on Wikipedia, is yielding the best recall in the identification of incorrect patches. There are several possible reasons for this situation: BERT implements the deepest neural network and builds on the largest training data. Its performance suggests to researchers that code-oriented embedding models should be trained on large code datasets or fine-tuned on specific target tasks in order to become competitive against BERT. While we were completing the experiments, a pre-trained CodeBERT [154] model has been released. In future work, we will investigate its relevance for producing embeddings that may yield higher performance in patch correctness prediction. In any case, we note that CC2Vec provided the best embeddings for yielding the best recall in identifying correct patches (using similarity thresholds). This finding suggests we use the embedding model built for code changes (e.g., CC2Vec) for the objective of having a high recall in identifying correct patches.

The small sizes of the code fragments lead to similar embeddings. Figure 3.11 illustrates the different cosine similarity scores that can be obtained for the BERT embeddings of different pairs of short sentences. Although the sentences are semantically (dis)similar, the cosine similarity scores are quite close. This explains why recalling correct patches based on a similarity threshold was a failed attempt ($\sim 5\%$ for APR-generated patches for Defects4J+Bears+Bugs.jar bugs). Nevertheless, experimental results demonstrated that deep learned features are relevant for learning to discriminate. Considering the different sizes of code fragments contained in each patch may affect the similarity computation, we suggest that researchers control the size of the code fragments of the patch when investigating the hypothesis in RQ-2 for patch correctness.

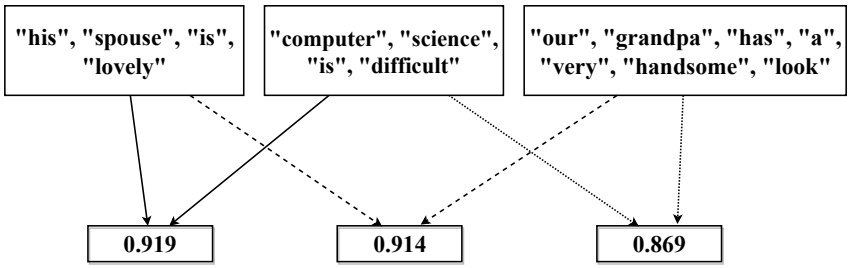


Figure 3.11: Close cosine similarity scores with small-sized inputs for BERT embedding model.

[*Embeddings are most suitable when applied to simple ML algorithms.*] Because embeddings are yielded from neural networks, they are actually formed by complex crossed features. When they are fed to a complex discriminant model such as Random Forrest, it may lead to overfitting with small datasets. Our experiments however show that simple Logistic Regression yields the best AUC, suggesting that this learner

was able to better identifying discriminating features for the prediction task.

3.5.2 Threats to validity

Our empirical study carries a number of threats to validity that we have tried to mitigate.

Threats to External Validity. There are a variety of representation learning models in the literature. A threat to validity of our study is that we may have a selection bias by considering only four embedding models. We have mitigated this threat by considering representative models in different scenarios (pre-trained vs retrained, code change specific vs natural language oriented). Another threat to validity is related to the use of Defects4J data in evaluating the ML classifiers. This choice however was dictated by the data available and the aim to compare against related work. Finally, with respect to the explored models, the attention system of CC2Vec requires some execution parameters to perform well. Since the relevant code was not available, we use use a non-attention version instead, potentially making CC2Vec embeddings be under-performing. We release the artifacts for future comparisons by the research community.

Threats to Internal Validity. A major threat to internal validity lies in the manual assessment heuristics that we applied to the RepairThemAll-generated dataset. We may have misclassified some patches due to mistakes or conservatism. This threat however holds for all APR work that relies on manual assessment. We mitigate this threat by following clear and reproducible decision criteria, and by further releasing our labelled datasets for the community to review.

Threats to Construct Validity. For our experiment, the considered embedding models are not perfect and they may have been under-trained for the prediction task that we envisioned. For this reason, the results that we have reported are likely an under-estimation of the capability of representation learning models to capture discriminative features for the prediction of patch correctness. Our future studies on representation learning will address this threat by considering different re-training experiments.

3.6 Related Work

Predicting Patch Correctness: To predict the correctness of patches, one of the first explored research directions relied on the idea of augmenting test inputs, i.e., more tests need to be proposed. Yang *et al.* [34] design a framework to detect overfitting patches. This framework leverages fuzz strategies on existing test cases in order to automatically generate new test inputs. In addition, it leverages additional oracles (i.e., memory-safety oracles) to improve the validation of APR-generated patches. In a contemporary study, Xin and Reiss [35] also explored to generate new test inputs, with the syntactic differences between the buggy code and its patched code, for validating the correctness of APR-generated patches. As complemented by Xiong *et al.* [2], they proposed to assess the patch correctness of APR systems by leveraging the automated generation of new test cases and measuring behavior similarity of the failing tests on buggy and patched programs.

Through an empirical investigation, Yu *et al.* [1] summarized two common overfitting issues: incomplete fixing and regression introduction. To assist alleviating the overfitting issue for synthesis-based APR systems, they further proposed `UnsatGuided` that relies on additional generated test cases to strengthen patch synthesis, and thus reduce the generation of incorrect overfitting patches.

Predicting patch correctness with thanks to an augmented set of test cases heavily relies on the quality of tests. In practice, tests with high coverage might be unavailable [148]. In our paper, we do not rely on any new test cases to assess patch correctness, but leverage representation learning techniques to build representation vectors for buggy and patched code of APR-generated patches.

To predict overfitting patches yielded by APR tools, Ye *et al.* [148] propose ODS, an overfitting detection system. ODS first statically extracts 4,199 code features at the AST level from the buggy code and generated patch code of APR-generated patches. Those features are fed into three machine learning algorithms (logistic regression, KNN, and random forest) to learn an ensemble probabilistic model for classifying and ranking potentially overfitting patches. To evaluate the performance of ODS, the authors considered 19,253 training samples and 713 testing samples from the Durieux *et al.* empirical study [28]. With these settings, ODS is capable of detecting 57% of overfitting patches. The ODS approach relates to our study since both leverage machine learning and static features. However, ODS only relies on manually identified features which may not generalize to other programming languages or even other datasets.

In a recent work, Csuvik *et al.* [125] exploit the textual and structural similarity between the buggy code and the APR-patched code with two representation learning models (BERT [155] and Doc2Vec [159]) by considering three patch code representation (i.e., source code, abstract syntax tree and identifiers). Their results show that the source code representation is likely to be more effective in correct patch identification than the other two representations, and the similarity-based patch validation can filter out incorrect patches for APR tools. However, to assess the performance of the approach, only 64 patches from QuixBugs [149] have been considered (including 14 in-the-lab bugs). This low number of considered patches raises questions about the generalization of the approach for fixing bugs in the wild. Moreover, unlike our study, new representation learning models (code2vec [167] and CC2Vec [163]) dedicated to code representation have not been exploited.

Representation Learning for Program Repair Tasks: In the literature, representation learning techniques have been widely explored to boost program repair tasks. Long and Rinard explored the topic of learning correct code for patch generation [38]. Their approach learns code transformation for three kinds of bugs from their related human-written patches. After mining the most recent 100 bug-fixing commits from each of the 500 most popular Java projects, Soto and Le Goues [150] have built a probabilistic model to predict bug fixes for program repair. To identify stable Linux patches, Hoang *et al.* [178] proposed a hierarchical deep learning-based method with features extracted from both commit messages and commit code. Liu *et al.* [166] and Bader *et al.* [179] proposed to learn recurring fix patterns from human-written patches and suggest fixes. Our paper is not aiming at proposing a new automated patch generation approach. We indeed rather focus on assessing representation learning techniques for predicting correctness of patches generated by program repair tools.

3.7 Conclusion

In this paper, we investigated the feasibility of statically predicting patch correctness by leveraging representation learning models and supervised learning algorithms. The objective is to provide insights for the APR research community towards improving the quality of repair candidates generated by APR tools. To that end, we, first investigated the use of different distributed representation learning to capture the similarity/dissimilarity between buggy and patched code fragments. These experiments gave similarity scores that substantially differ for across embedding models such as BERT, Doc2Vec, code2vec and CC2Vec. Building on these results and in order to guide the exploitation of code embeddings in program repair pipelines, we investigated in subsequent experiments the selection of cut-off similarity scores to decide which APR-generated patches are likely incorrect. This allowed us to filter out between 31.5% and 94.9% incorrect patches based on brute cosine similarity scores. Finally, we investigated the discriminative power of the deep learned features by training machine learning classifiers to predict correct Patches. Decision Tree, Logistic Regression and Naive Bayes are tried with code embeddings from BERT, Doc2Vec and CC2Vec. Logistic Regression with BERT embeddings yielded very promising performance on patch correctness prediction with metrics like F-Measure at 0.72% and AUC at 0.8% on a labeled deduplicated dataset of 1000 patches. We further showed that the performance of these models on static features is promising when compared to the state of the art (PATCH-SIM [2]), which uses dynamic execution traces. Experimental results suggest that the deep learned features can be complementary to hand-crafted features (such as those engineered by ODS [148]). This finding underscores the efficacy of employing representation learning techniques to assess the behavior of code changes in relation to patch correctness. It also encourages further exploration into the viability of combining learned and engineered features for accurate prediction.

Availability. All artifacts of this study are available in the following public repository:

`https://github.com/SerVal-DTF/DL4PatchCorrectness`

4 Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches

In this chapter, we propose to investigate the combination of deep learned embeddings and engineered features for the accurate prediction of patch correctness. By combining deep learned embeddings and engineered features, our proposed approach PANTHER *outperforms the previous state of the arts with higher scores in terms of AUC, +Recall and -Recall, and can accurately identify more (in)correct patches that cannot be predicted by the classifiers only with learned embeddings or engineered features. Finally, we use an explainable ML technique, SHAP, to empirically interpret how the learned embeddings and engineered features are contributed to the patch correctness prediction.*

This chapter is based on the work published in the following research paper:

- **Haoye Tian**, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F. Bissyandé. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. *ACM Transactions on Software Engineering and Methodology* 32, no. 4 (2023): 1-34.

Contents

4.1 Overview

The issue of overfitting patches represents a significant obstacle in generate-and-validate APR strategies [145, 50, 132]. Recent research on APR systems underscores the crucial need for an accurate estimation of the proportion of valid patches that can be discovered. In order to enhance this proportion, various directions have been explored by researchers [34, 35, 46, 38]. So far, the state-of-the-art works targeting the identification of patch correctness are based on computing the similarity of test case execution traces [2], or using machine learning to identify correct patches based on engineered static code features [148], pre-trained natural language-based embeddings [125], and source code trained embeddings [3].

This paper. In this work, we extensively study and evaluate how effective are source code embeddings and engineered features in predicting correct patches. For example, which set of features: engineered or learned embeddings yield better performance in predicting correct patches? Can a combination of both kinds of feature achieve higher performance? Our work fills this gap.

This work builds on and extends the research conducted in Chapter 3 in the following manner:

- We examine and compare the effectiveness of *code embeddings*, *engineered features*, and their combination for predicting patch correctness.
- We present an analysis for detecting which kinds of features contribute to the (in)correct prediction of patch correctness.

We investigate in this paper the feasibility of leveraging advances in deep representation learning to produce embeddings for APR-generated patches and their engineered features, that are amenable to reasoning about correctness. The main contributions can be summarized as follows:

❶ We evaluate our proposed approach Leopard and state of the art approaches by applying a 10-group cross validation in a practical perspective. Comparing against the state of the art, Leopard is complementary to them, even outperforms them on filtering out incorrect patches.

❷ We explore the combination of the learned embeddings and the engineered features to improve the performance on identifying patch correctness with more accurate classification, and implement an upgraded version of Leopard, that we named Panther. The exploring examination is supported by our experimental results.

❸ We empirically interpret the cause of prediction behind features and classifiers to help aware the essence of identifying patch correctness with an explainable ML technique SHAP.

4.2 Background

This works leverages engineered features and machine learning techniques to tackle the problem of identifying correct patches among incorrect and plausible APR-generated patches. Additionally, we examine the explainability of ML models used to predict correct patches. The explainability aspect is of high importance to developers applying APR in their workflow. Therefore, we begin by providing the necessary background of the two pillars of our work: (i) engineered features for predicting patch correctness, (ii) the explainability of ML models using SHAP.

4.2.1 Engineered Features

Engineered features are carefully designed and selected features which represent and capture important properties of the underlying data. In APR, one possibility is to statically extract those features from the abstract syntax tree (AST) of the buggy code, the AST of the patched path and the related AST edit scripts as proposed by ODS [148].

ODS extract three kinds of features to detect correct patches: (i) Code description features, e.g., kinds of specific operators in patch code and kinds of statements, (ii) Repair pattern features, whether the repair code has specific patterns according to [180], and (iii) Contextual syntactic features, e.g., the types of faulty statements and the types of their surrounding statements. Using these engineered features, ODS trains a series of machine learning classifiers to predict patch correctness. The experimental evaluation on 713 patches shows that ODS can filter out 57% of overfitting patches and exhibits competitive results when compared with state of the art. We adopt ODS engineered features to conduct our study. Because ODS can not steadily generate contextual syntactic features for our patches, we consider mainly using, in our study, two rest kinds of engineered features[1]: (1) Code description features and (2) Repair pattern features.

4.2.2 SHAP - SHapley Additive exPlanations

SHAP is a unified framework proposed by Lundberg *et al.* [181] to interpret the output of machine learning models. It connects optimal credit allocation with local explanations using the classic Shapley values from the game theory and their related extensions, thus can provide the importance of each feature for certain particular prediction. Through SHAP, the positive and negative effect of features on prediction can be generated, which allow practitioners to understand which behaviors lead to the (in)correct prediction. Besides, SHAP provides the interaction analysis between features to explore how different features are complementary to each other.

[1]We have received confirmation from the authors about this bug and the effectiveness of these two kinds of features.

4.3 Methodology

In this section, we first present the methodology of our study and then we introduce the research questions that we aim to answer using the proposed methodology.

Overall, our goal is to study the effectiveness of different representations of APR-generated patches and codes for the task of predicting which patches are correct. We first investigate a widespread hypothesis that a patch incurring minimal changes is more likely to be correct. To quantify the patch changes, we exploit different code representation learning methods that leverage deep learning techniques to learn features for code. We adapt them to generate the vectors of buggy code and patched code as well as compute the similarity value of vectors. Based on the similarity distribution, we experimentally filter out incorrect APR-generated patches by relying on naively-defined thresholds.

In the view of learning representation reveals the properties of code related to patch correctness, we propose to further identify patch by training classifiers (learners) on the representation vector of a patch. Figure 4.1 provides an overview of such a pipeline and its variants. To represent patches in a format suitable for learning algorithms, we use the aforementioned representation learning methods to generate vectors for buggy code and patched code. Afterwards, we cross the vectors by applying subtraction, multiplication, cosine similarity and euclidean similarity to obtain the deep learned feature of the patches. The resulting patch embedding has 2*n+2 dimensions where n is the dimension of input code fragment embeddings. The values of the dimension n for BERT, Doc2Vec and CC2Vec are set as 1024, 64 and 64, respectively. On the other hand, we also exploit the manually engineered features that are extracted from the given data, the patch in our case, and aim to capture specific information that is thought to be relevant to the patch correctness. The dimension m for ODS is 195.

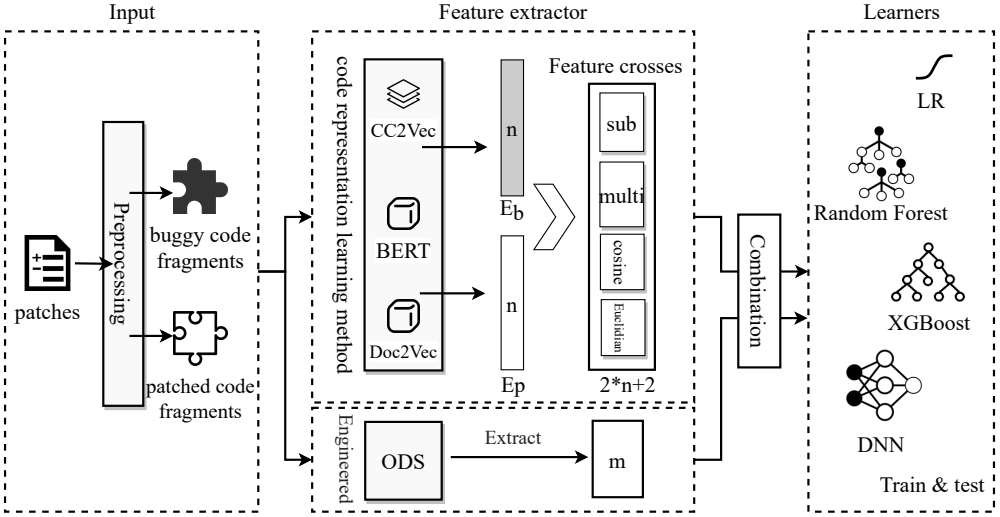


Figure 4.1: Overview of PANTHER.

Learned and engineered features represent a patch from different perspectives. To improve the identification performance of patch correctness, we further propose three methods (*Ensemble learning*, *Naïve Vector Concatenation*, and *Deep Combination.*) to combine the two features for obtaining the informative representation of a patch. After obtaining a vector that represents a given patch, different machine learning algorithms such as random forest or a deep neural network (DNN) are trained as

classifiers that distinguish correct from incorrect APR patches. In the end, we provide the SHAP explanation of the features and interaction of different features that contribute to the patch correctness prediction. In the following, we present the details of each research question.

Research Questions

RQ-1: *Can we learn to identify patch correctness by training predictors with learned embeddings and engineered features of code input in realistic scenario?* Through cross-project evaluation (10-group cross validation), we investigate whether such a machine learning predictor built with static features can provide comparable performance with dynamic approaches, such as PATCH-SIM, which leverage execution behaviour information. We also compare the performance yielded when using deep learned features against the performance yielded when using the engineered features in the state of the art.

RQ-2: *Can the combination of learned embeddings and engineered features achieve optimum performance for predicting patch correctness?* We investigate the possibility of ensuring high accuracy in patch correctness identification by combining different representations of patches.

RQ-3: *Which features are most useful for predicting patch correctness?* We leverage SHAP explanation models to provide an interpretation of the contribution of different features to the predictions.

4.4 Experiments and Results

We first introduce the metrics used in the experiments. Then, we present the experiments that we designed to answer the research questions of our study. For each experiment, we state the objective, overview the execution details, and present the results.

Our objective is to measure the ability of the approaches in terms of recalling correct patches while filtering out incorrect patches. Thus, we use the definitions of **Recall** for the evaluation of the patch correct assessment systems:

- **+Recall** measures to what extent correct patches are identified, i.e., the percentage of correct patches that are identified from all correct patches.
- **-Recall** measures to what extent incorrect patches are filtered out, i.e., the percentage of incorrect patches that are filtered out from all incorrect patches.

$$+Recall = \frac{TP}{TP + FN} \qquad (4.1) \qquad\qquad -Recall = \frac{TN}{TN + FP} \qquad (4.2)$$

where TP represents true positive, FN represents false negative, FP represents false positive, TN represents true negative.

Accuracy and Precision. The ratio of positive and negative samples of our dataset is balanced (1.3:1). We thus use accuracy and precision to evaluate the performance of the approaches in classifying the patches.

Area Under Curve (AUC) and F1-measure. We train a few machine and deep learning-based classifiers to identify the patch correctness. Therefore, we use two commonly used metrics for evaluating overall performance of the classifiers: AUC (the overall ability to distinguish between correct and incorrect patches) and F1 score (harmonic mean between precision and recall for identifying correct patches).

4.4.1 RQ-1: Classification of Correct Patches with Supervised Learning

[Objective]: Cosine similarity between learned embeddings (which was used in the previous experiments) considers every deep learned feature as having the same weight as the others in the embedding vector. We investigate the feasibility to infer, using machine learning, the weights that different features may present with respect to patch correctness. To this end, we build a patch correctness prediction framework, LEOPARD (LEarn tO Predict pAtch coRrectness with embeDdings), with the embedding models and machine learning algorithms. We compare the prediction evaluation results of LEOPARD with the achievements of related approaches in the literature. The experiments are performed towards providing insights for the three sub-questions:

- RQ-1.1 *Can LEOPARD learn to predict patch correctness by training classifiers based on the learned embeddings of code ?*
- RQ-1.2 *Can LEOPARD be as reliable as a dynamic state-of-the-art approach such as PATCH-SIM in the patch correctness identification task?*
- RQ-1.3 *To what extent learned embeddings of LEOPARD are providing different prediction results than the engineered features?*

[Experimental Design for RQ-1.1]: To perform our machine learning experiments, we first require a ground-truth dataset. To that end, we rely on labeled datasets in the literature. Since incorrect patches generated by APR tools are only available for the Defects4J bugs, we focus on labeled patches provided by three

Table 4.1: Dataset for evaluating ML-based predictors of patch correctness.

	Correct patches	Incorrect patches	Total
Liu *et al.* [175]	94	366	460
Ye *et al.* [182]	242	452	694
Xiong *et al.* [2]	30	109	139
Defects4J (developers) [88]	969	0	969
Other APR tools	263	162	425
Dataset	1,598	1,089	2,687
Dataset (deduplicated)	1,288	956	2,244
Dataset (final, with available features)	**1,199**	**948**	**2,147**

independent teams (Liu *et al.* [175], Ye *et al.* [182] and Xiong *et al.* [2]) and other patches generated by APR tools. Very few patches generated by the different tools are actually labeled as correct, which leads to an imbalanced dataset. To reduce the imbalance issue, we supplement the dataset with developer (correct) patches as supplied in the Defects4J benchmark. Note that one developer patch could include multiple fixing hunks for different files, but the extraction of engineered features only work on the patches with respect to changing single file. Thus, we split such patches into sub patches by their changed files to ensure that one sub patch is only involved with one code file. In total, we collect 2,687 patches. After removing duplicates, 2,244 patches are remained. 97 patches are failed to obtain their engineered feature. Eventually, the ground-truth dataset is built with 2,147 patches, shown in Table 4.1.

Our ground truth dataset patches are then fed to our embedding models in Leopard to produce embedding vectors. As for previous experiments, the parsability of Defects4J patch code fragments prevented the application of code2vec: Leopard uses pre-trained models of BERT (trained with natural language text) and CC2Vec (trained with code changes) as well as a retrained model of Doc2Vec (trained with patches). Since the representation learning models are applied to code fragments inferred from patches (and not to the patch themselves), Leopard collects the embeddings of both buggy code fragment and patched code fragment for each patch. Then Leopard must merge these vectors back into a single input vector for the classification algorithm. We follow an approach that was demonstrated by Hoang et al. [163] in a recent work on bug fix patch prediction: the classification model performs best when features of patched code fragment and buggy code fragment are crossed together.

At first, and following related works in the literature, we used a 10-fold cross validation scheme to evaluate and compare our approach against the state of the art. However, we found that, with this scheme, a patch set generated for the same bug can be split into both the training and testing sets. Such a scenario is actually unrealistic (and biased) since we should not train the model with some labeled patches of a bug that we intend to repair (test set). To address this bias, we propose instead a 10-group cross validation scheme: First, we randomly distribute all bugs into 10 groups. Every group contains unique bugs and their associated patches. Then, we use 9 groups as train data and the remaining group as the test data. Finally, we repeat the selection of train and test groups for ten rounds and obtain the average score of the metrics.

[Experimental Results for RQ-1.1]: We compare the performance of different embedding models using different classification algorithms. Table 4.2 presents the results with 10-group cross validation setup. All classical metrics used for assessing predictors are reported: Accuracy, Precision, Recall, F1-Measure, Area Under Curve

Table 4.2: Evaluation of learned embeddings on six ML classifiers in LEOPARD.

Learner	Embedding	Accuracy	Precision	Recall	F1-measure	AUC
Decision Trees	BERT	62.1	64.7	70.8	67.6	0.611
	CC2Vec	58.0	61.7	65.5	63.5	0.572
	Doc2Vec	58.7	62.0	67.6	64.6	0.576
Logistic regression	BERT	72.2	73.5	78.7	76.0	0.796
	CC2Vec	61.8	64.8	68.9	66.8	0.679
	Doc2Vec	65.8	66.6	77.7	71.7	0.717
Naive bayes	BERT	66.5	72.5	57.6	65.7	0.726
	CC2Vec	57.6	70.1	31.9	45.7	0.670
	Doc2Vec	55.9	63.0	51.0	56.4	0.610
Random forest	BERT	69.4	68.3	77.9	75.5	0.793
	CC2Vec	62.1	63.9	74.1	68.6	0.705
	Doc2Vec	64.9	63.5	87.6	73.6	0.705
XGBoost	BERT	71.8	71.6	82.1	76.5	0.803
	CC2Vec	65.3	66.4	76.6	71.1	0.729
	Doc2Vec	63.2	63.5	80.2	70.8	0.693
DNN	BERT	70.3	74.4	71.3	72.8	0.767
	CC2Vec	51.8	55.5	69.0	61.6	0.503
	Doc2Vec	63.2	64.7	75.1	69.5	0.679

(AUC). XGBoost applied to BERT embeddings yields the best performance on the most of metrics (e.g. AUC with 0.803 and F1-measure with 0.765), while DNN achieves the best performance on precision of 0.744.

Our previous work 3 was conducted through a 5-fold cross validation. To evaluate performance change of the approach on the new augmented dataset, we re-conduct a 5-fold cross validation experiment. The results show that after increasing the number of training examples (1,147 more patches), the performance of the decision tree, logistic regression and naive bayes classifiers are improved. For instance, applying the three classifiers with BERT embeddings, their accuracy, precision, recall and F1-measure are improved with 3 to 23.6 points (except the recall of Naive bayes + BERT embedding is decreased). Their AUC values are increased with 0.067, 0.06, 0.126, respectively. These results provide us the possibility of evolving the patch identification through datasets augmentation. Note that, for the following experiment, we proceed to focus on using 10-group cross validation because of its effectiveness for evaluating the approaches in practice.

✍ **RQ3.1** ▶ *Tree-based boosting classifiers (Random forest and XGBoost) and Deep learning classifier (DNN) with BERT embeddings yield the promising performance on predicting the patch correctness for APR tools (e.g., F1-measure at 76.5% and AUC at 80.3%).*◀

[Experimental Design for RQ-1.2]: PATCH-SIM [2] is the state-of-the-art work on predicting the patch correctness for APR tools. It is a dynamic-based approach, which generates execution traces of patched programs with new generated tests, and compares the execution traces across test cases to assess the correctness of APR-generated patches. We propose to apply PATCH-SIM to our collected patches (cf. Table 4.1). Unfortunately, PATCH-SIM is implemented to run on Defects4J-v1.2.0[2]. Therefore, it failed to process 476 patches generated for some bugs (e.g., JSoup bugs) in the latest version of Defects4J (i.e., Defects4J-v2.0.0). Furthermore, even when PATCH-SIM can run, we observe that it does not yield any prediction

[2] `https://github.com/rjust/defects4j/releases/tag/v1.2.0`

output for 1,022 patches[3]. Eventually, we were able to assess the performance of PATCH-SIM on 649 patches. To avoid a potential bias in comparisons, we also conduct the ML-based classification experiments for LEOPARD on the 649 patches.

[Experimental Results for RQ-1.2]: Table 4.3 provides the comparing results on predicting patch correctness. In terms of Recall, PATCH-SIM achieved 78.9% that is a bit higher than the BERT embedding + Random forest of LEOPARD, which demonstrates its ability of recalling correct patch from plausible patches as reported in [2] by its authors. However, the accuracy, precision and AUC measurements are just 38.8%, 24.7% and 52.8%, respectively. These results underperform the three ML classifiers of LEOPARD. It indicates the many incorrect patches are wrongly identified as correct by PATCH-SIM. Figure 4.2 further gives an example on comparing the BERT embedding + the XGBoost classifier of LEOPARD and PATCH-SIM in terms of the number of (in) patches correctly identified by them. XGBoost classifier of LEOPARD can recall more correct and incorrect patches than the PATCH-SIM, and the 24 correct patches and 124 incorrect patches are exclusively correctly predicted by it.

Table 4.3: Comparing evaluation of LEOPARD (BERT embedding + ML classifiers) against PATCH-SIM.

Approach		Accuracy	Precision	Recall	F1-measure	AUC
PATCH-SIM		38.8	24.7	78.9	37.7	0.528
LEOPARD	BERT + Random forest	41.3	25.5	78.3	38.4	0.594
	BERT + XGBoost	**42.7**	**26.2**	79.6	39.4	**0.614**
	BERT + DNN	40.0	26.1	**85.5**	**40.0**	0.546

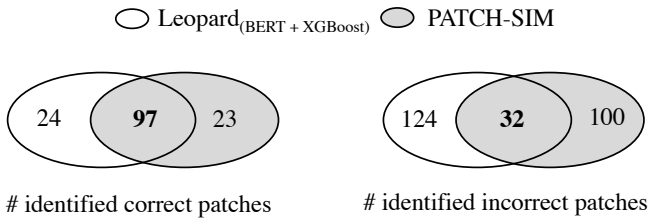


Figure 4.2: Comparison on the number of (in)patches correctly identified by LEOPARD (with the BERT embeddings + the XGBoost learner) against PATCH-SIM.

Time cost. Note that we have recorded that, on average, PATCH-SIM takes ∼17.5 minutes to predict the correctness of each patch. In contrast, each of the ML classifiers of LEOPARD takes less than 1 minute for prediction. However, note that the training of LEOPARD requires the input of the learned embeddings of patches generated by pre-trained models (e.g. BERT). Such models, which are available on-the-shelf, have been trained using hundreds of TPUs that were run for several hours on a large corpus.

✍ **RQ-1.2** ▶ *ML predictors of LEOPARD trained on learned embeddings can be complementary to the state-of-the-art PATCH-SIM. They can also outperform PATCH-SIM in filtering out more patches generated by APR tools.*

Experimental Design for RQ-1.3: As reported by Ye *et al.* [148] in a recent

[3]We have reported the issue to the authors but have not yet been made aware of any solution to address it. Note that in their original paper the authors transparently informed readers that the tool indeed is sensitive to the datasets.

Table 4.4: Evaluation of engineered feature on six ML classifiers.

Learner	Accuracy	Precision	Recall	F1-measure	AUC
DecisionTree	66.6	68.6	73.9	71.1	0.666
Logistic regression	70.0	72.7	74.1	73.4	0.773
Naive bayes	49.6	74.6	14.7	24.5	0.689
Random forest	70.7	72.1	77.5	74.7	0.769
XGBoost	70.5	72.6	79.9	74.1	0.776
DNN	69.8	72.1	74.8	73.4	0.777

study, post-processing APR-generated patches through engineered features achieves promising results. Therefore, in this study, we also use some of the engineered features (Prophet features and repair pattern) in [148] to predict correct patches on a larger dataset: overall, our study is based on 2,147 patches while Ye *et al.* applied only 713 patches. Results in this study are given based on 10-group cross validation.

Results for RQ-1.3: Table 4.4 presents the results of predicting patch correctness with the engineered features. The naive bayes learning algorithm achieves a unusual performance compared to the other five learners. It yields the highest precision, but leads to a much lower recall than others. This suggests that a very small number of correct patches can be recalled via using this learner. The Random Forest and XGBoost learners achieve similarly high performance (e.g., F1-measure at 74.7%/74.1% and AUC at 76.9%/77.6%), and are followed by the DNN learner. Overall, the performance reached with engineered features is generally comparable (in terms of global metrics) to that yielded by Leopard using learned embeddings, except when using the Naive Bayes and Decision Trees learning algorithm.

Figure 4.3 further illustrates the differences between the XGBoost classifier with the BERT embeddings and the engineered features in terms of the number of identified (in)correct patches. More (in)correct patches can be correctly identified by the XGBoost classifier with both two scenarios. Nevertheless, there still is a big complementary space of identifying the patch correctness for the two scenarios.

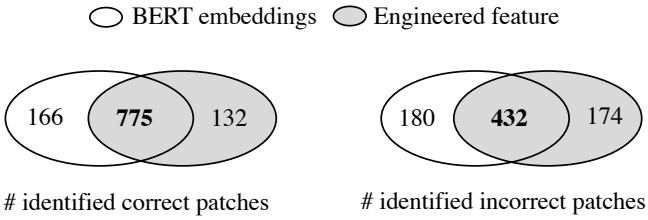


Figure 4.3: Comparison on the number of (in)patches correctly identified by the XGBoost classifier with the BERT embeddings and the engineered features.

✍ **RQ-1.3** ▶ *The ML classifiers fed with the engineered features (from static code) can achieve comparable performance to learned embeddings based classifiers in identifying patch correctness. There is nevertheless the possibility to improve the prediction performance in both cases since their correct predictions are not perfectly overlapping: learned embeddings lead to the identification of correct/incorrect patches that are not recalled with engineered features and vice versa.* ◀

4.4.2 RQ-2: Combining Learned Embeddings and Engineered Features for more Accurate Classification of Correct Patches

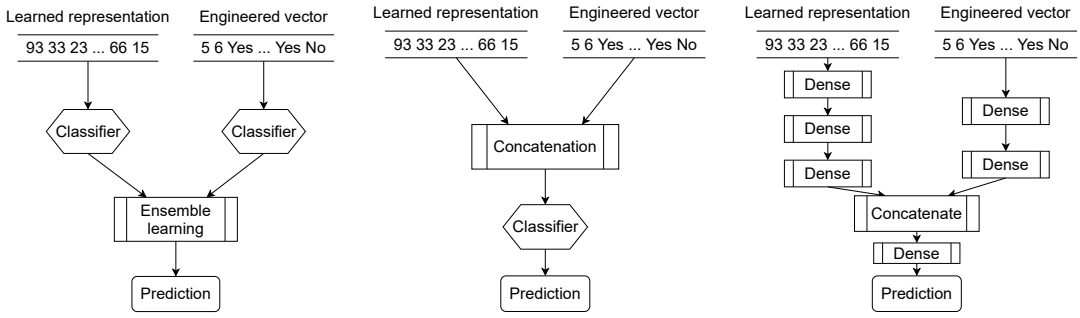
[**Objective**]: Following up on the insights from the previous research question, which compared engineered features against learned embeddings, we investigate the potential of leveraging both feature sets to improve the classification of correct patches.

[**Experimental Design**]: Leveraging different feature sets can be achieved in several ways, e.g., by concatenating feature vectors or by performing ensemble learning. In this study, we investigate three different methods which are implemented in the upgraded version of LEOPARD, PANTHER (Predict pAtch correctNess wiTH the learned Embbeddings and engineeRed features), as illustrated in Figure 4.4:

1. *Ensemble learning.* We rely on the six learning algorithms (cf. Tables 4.2 and 4.4) to predict the correctness of patches based either on the learned embeddings or on the engineered features. Eventually, to combine both, we simply compute the average prediction probability provided by a pair of classifiers (one trained with learned embeddings and the other with engineered features), and use this probability to decide on patch correctness.
2. *Naïve Vector Concatenation.* In the second method, we ignore the fact that learned embeddings vectors and engineered feature vectors are not from the same space and propose to Naïvely concatenate them into a single representation. Our intuition, indeed, is that both representations capture different features of patches and can therefore offer, together, a better representation. The yielded concatenated vectors are then used to train the classifiers (with the usual learning algorithms).
3. *Deep Combination.* In the last method, we consider that learned embeddings and engineered features are from different spaces. Therefore, we must learn their different weights as well as the common representations for them before concatenation. We resort thus to deep neural networks to attempt a deep combination of feature sets before classification.

In this RQ, given the performance of BERT in previous experiments (cf. Table 4.2), we focus on the BERT embedding model to learn the learned embeddings of patches. Similarly, we only consider Random forest and XGBoost as the best learners to be applied (cf. Table 4.2 and Table 4.4). The *Deep Combination* method is based on the work of Cheng *et al.* [183] who proposed a deep learning fusion structure which combined layers that were specialized to explore memorization and generalization of features. Following up this idea of fusion, we design a Double-DNN-fusion structure where learned embeddings are considered useful for generalization and engineered features are considered for memorization. Eventually, we conduct 10-group cross validation for the experimental assessment.

[**Experimental Results**]: Table 4.5 presents the performance comparison for correctness identification when using combined features vs using single feature sets. The comparison is done in terms of three main metrics: +Recall (to what extent correct patches can be identified), -Recall (to what extent incorrect patches can be filtered out), and AUC (area under the ROC curve, i.e. comprehensive performance of the predictor). Overall, the performance of classifying correct patches is improved after using each of the three combination strategies (except the -Recall of the random forest classifier with the *Naïve Vector Concatenation*) for the learned (BERT) and



(a) Ensemble learning. (b) Naïve Vector Concatenation. (c) Deep Combination.

Figure 4.4: Combination options of features for patch classification in PANTHER.

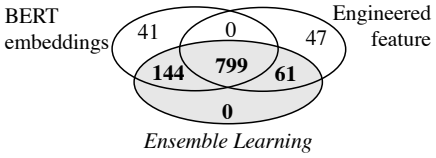
Table 4.5: Comparing results of classifying correct patches with combined feature against the single feature.

Tool	Feature	Accuracy	Precision	+Recall	-Recall	F1-measure	AUC
			Random Forest				
LEOPARD	BERT embeddings	0.694	0.683	0.779	0.624	0.755	0.793
	Engineered feature	0.707	0.721	0.775	0.620	0.747	0.769
PANTHER	*Ensemble Learning*	0.745	0.740	**0.837**	**0.629**	0.786	**0.818**
	Naïve Vector Concatenation	0.708	0.693	0.786	0.629	0.766	0.799
			XGBoost				
LEOPARD	BERT embeddings	0.718	0.716	0.821	0.588	0.765	0.803
	Engineered feature	0.705	0.726	0.799	0.596	0.741	0.776
PANTHER	*Ensemble Learning*	0.757	0.754	**0.837**	**0.655**	0.794	**0.822**
	Naïve Vector Concatenation	0.730	0.725	0.833	0.600	0.775	0.811
			DNN				
LEOPARD	BERT embeddings	0.703	0.744	0.713	0.690	0.728	0.767
	Engineered feature	0.698	0.721	0.748	0.634	0.734	0.777
PANTHER	*Deep Combination*	0.730	0.760	**0.757**	**0.696**	0.758	**0.798**

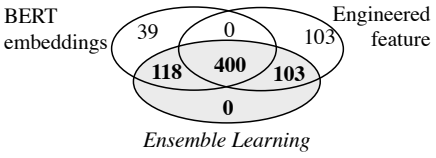
engineered (ODS) feature. With respect to **+Recall** (i.e., recalling the correct patches), the Random forest and XGBoost based classifier with *Ensemble Learning* achieve the highest value at 83.7%, improving by 1 to 6 percentage points the performance with single feature sets. With respect to **-Recall** (i.e., filtering out the incorrect patches), the best classifier is DNN-based with the *Deep Combination* of features: it achieves the highest recall in correctly excluding 69.6% of the incorrect patches. With respect to **AUC**, the XGBoost-based classifier with the *Ensemble Learning* present the best performance at 82.2%, improving by 2 to 5 percentage points the performance with single feature sets. To sum up, combining the BERT embeddings of patches with their ODS features does improve the performance of identifying patch correctness. Note that the results show that, in general, Ensemble Learning applied to independently trained classifiers yields the highest performance gains. The McNemar's statistical hypothesis test [184] further confirms that the gains are statistically significant for the *Ensemble Learning* and *Deep Combination* while it is not the case for the *Naïve Vector Concatenation*. This suggest that the features (learned and engineered) are from different spaces and are best exploited when applied standalone to model patch correctness, and can complement each other in terms of prediction.

Figure 4.5 further highlights the number of (in)correct patches identified based on BERT embeddings, engineered features and the combined features, respectively. Since the "Random forest" learner presents a similar performance with "XGBoost", Figure 4.5 focuses on the latter.

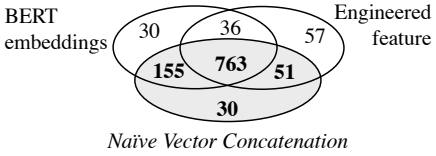
From a **qualitative point of view**, with the *Ensemble Learning*, more (in)correct



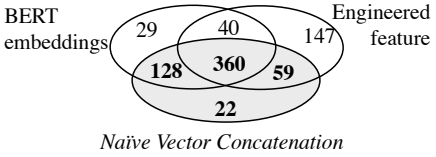
(a) Identified correct patches.



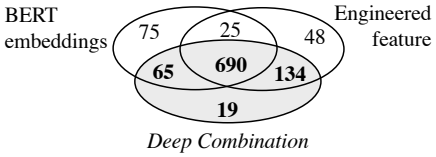
(b) Identified incorrect patches



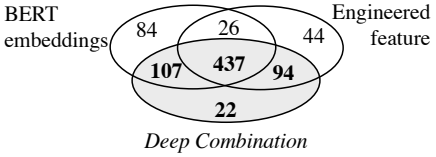
(c) Identified correct patches.



(d) Identified incorrect patches.



(e) Identified correct patches.



(f) Identified incorrect patches.

Figure 4.5: Comparison on the number of patches identified with the combined feature vs. the simple feature.

patches can be identified than each single feature set (i.e., BERT embeddings or engineered features). However, this combination does not help to identifying patches that were not identified using at least one feature set. In contrast, with *Naïve Vector Concatenation* and the *Deep Combination*, which combine features before classification, we can identify some (in)correct patches that could not be identified using either feature set alone.

From a **quantitative point of view**, the *Naïve Vector Concatenation* helps to identify slightly more correct patches (among those that could not be identified by each feature set alone) than the *Deep Combination*. As for new identified incorrect patches, they achieve the same metrics. Nevertheless, overall, the *Ensemble Learning* method helps to identify more correct patches while the *Deep Combination* helps to identify more incorrect patches.

✍ **RQ-2** ▶ *Leveraging learned embeddings (BERT) and engineered features (ODS) contributes to improve the performance in predicting patch correctness for APR tools. Merging independently trained classifiers achieves higher performance compared to each separate classifier, but does not lead to the identification of correct/incorrect patches that could not be identified by at least one of the classifier. In contrast, feature combination (i.e., Naïve Vector Concatenation and Deep Combination) before classification training appears to provide more information to discriminate some patches that were not correctly classified based on their learned embeddings or their engineered features alone.* ◀

4.4.3 RQ-3: Explanation of Improvements of Combination

[Objective]: The experimental results for previous RQs show that ML classifiers built based on learned embeddings, or on engineered features, or on both, yield promising performance in predicting patch correctness. The fact remains, however,

that the classifier is a black box model for practitioners. In particular, when leveraging combined feature sets, it may be helpful to investigate the impact of different features on the identification of patch correctness. To that end, we propose to build on Explainable ML techniques to explore how the models are built. In this work, we focus on Shapley Values, which compute the contributions of each feature in a given prediction. Shapley values originate from the field of game theory and have been implemented in the SHAP framework [181], which is widely used in the AI community.

[Experimental Design]: Our experiments are focused on the classifier yielded with the *Naïve Vector Concatenation* method since it managed to recall more correct patches through combining learned embeddings and engineered features (cf. RQ-3.3 in Section 4.4.1). We consider the case where the classifier is trained with the XGBoost learning algorithm. Using SHAP values as metric of feature importance, we investigate the top most important features that contribute to the combined model predictions. We further compare those important features against the features that are most contributing when the classifier is trained only with learned embeddings or only with engineered features. Finally, we present three specific patches that identified by different feature sets to observe the contribution of the features to prediction.

[Experimental Results]: Figure 4.6 illustrates the top-10 most contributing features: a feature named B-i refers to the ith feature learned with BERT. Others (e.g., *singleLine* and *codeMove*) refer to engineered features. The appearance of features from learned and engineered feature sets among the most contributing features suggests that both types of features are not only relevant but are also exploited in the yielded classifier.

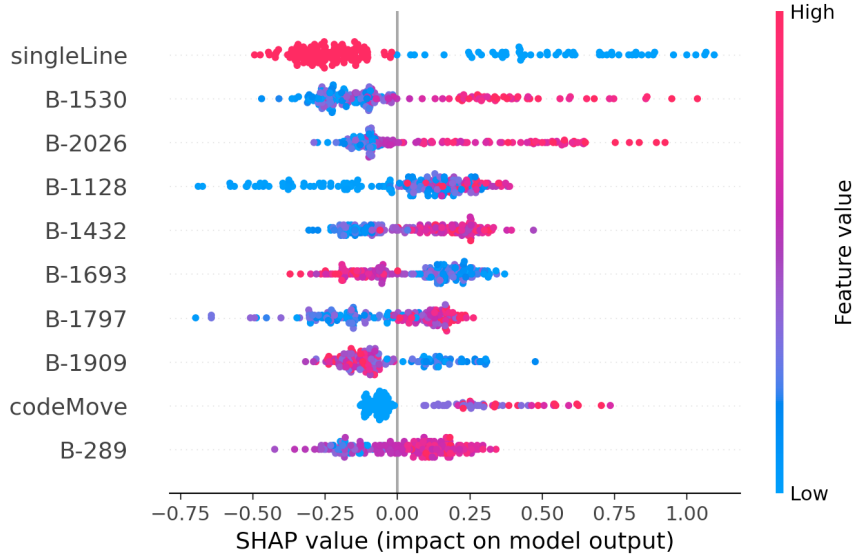


Figure 4.6: Top-10 Contributing Features (based on SHAP values) for the Classifier built by combining learned embeddings and engineered features.

Reading a SHAP explanation graph: In a given SHAP graph, each row is a distribution values for a given feature (Y-axis), where each data point is associated to one sample input data (i.e., a patch in our case). The color indicates the feature value, which is normalized: the more red, the higher the value. The X-axis represents the SHAP values, which indicate to what extent a given feature impacted the model output for a given patch. For example, most patches with high value (red) for feature

singleLine are located on the left (negative SHAP value), which suggests negative impact of *singleLine* on correctness prediction. It should be noted that, eventually, it is the contributions of different features that will be merged to yield the final prediction for each sample.

In Figure 4.6, we note that *singleLine* and *codeMove* are the top contributing engineered features among the combined feature sets. As we see from the figure, their red (high value) points and blue (low value) points are clearly separated to two sides, which demonstrates their values have obvious positive or negative effects on the model output. In Figure 4.7, when leveraging only engineered features, *singleLine* and *codeMove* also have significant contributions and are appearing in the 1st and 4th positions among the top contributing features. This indicates that the engineered features must be high-contributors to the decision (e.g., in terms of information gain) as shown in Figure 4.7, in order to obtain an efficient combination with learned features. Therefore, in practice we suggest that the research community should focus more on devising few but effective engineered features instead of massive but inefficient features to improve the performance of models.

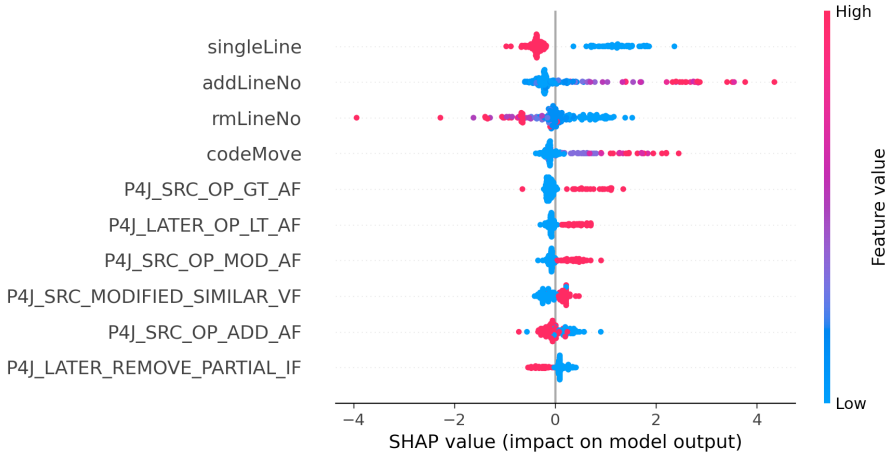
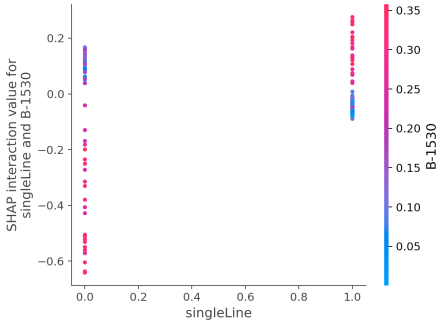


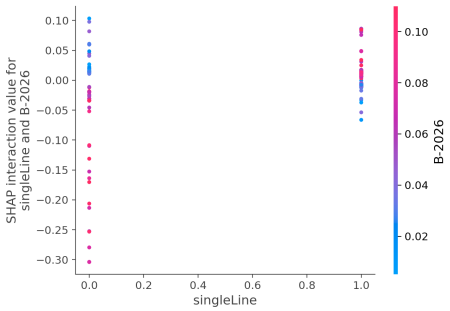
Figure 4.7: Top-10 contributing features (based on SHAP values) for the Classifier built only by the engineered features.

Overall, the SHAP explanations suggest that engineered features have an important effect on model prediction (because they appear among the top contributing features) but are complementary to the learned feature set. Indeed, the combination with *Naive Vector Concatenation* enables classifiers to identify correct patches that could not be identified when each feature set was used without the other. Therefore, we conclude that it is the interaction among the features that yields such a performance improvement. We propose to further investigate the interaction among pairs of features (one from the engineered features set and the other from the learned features set).

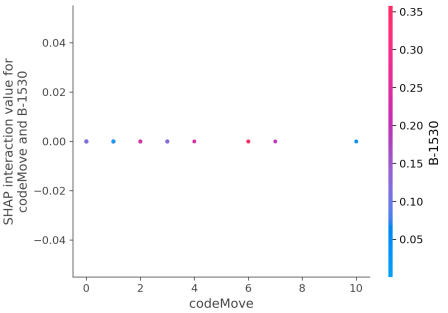
Figure 4.8 illustrates the interactions information provided by SHAP among *singleLine*, *codeMove* and *B-1530*. As it can be seen, in Figure 4.8(a), when the feature value of *singleLine* is 0, higher (redder) feature values of *B-1530* will lead to a more negative SHAP value for *singleLine* (i.e., it has negative impact on patch correctness prediction). In contrast, when the feature value of *singleLine* is 1, the same higher feature values of *B-1530* will tend to draw a positive SHAP value (i.e., positive impact). This example illustrates how learned and engineered features can interact to balance their contributions for the final predictions based on their respective feature values. Figure 4.8(b) and Figure 4.8(d) exhibit effective interaction



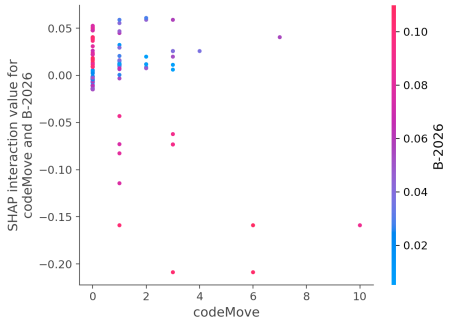
(a) Interaction between singleLine and B-1530.



(b) Interaction between singleLine and B-2026.

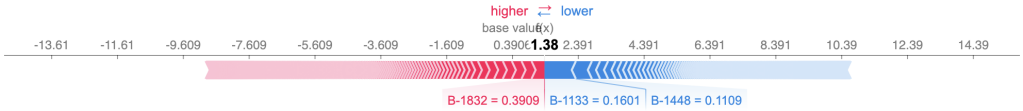


(c) Interaction between codeMove and B-1530.

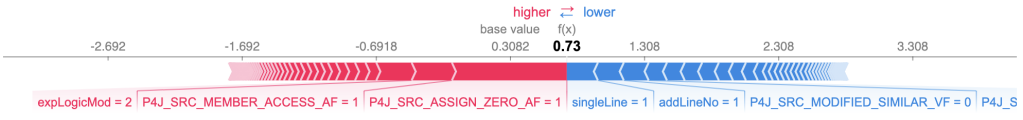


(d) Interaction between codeMove and B-2026.

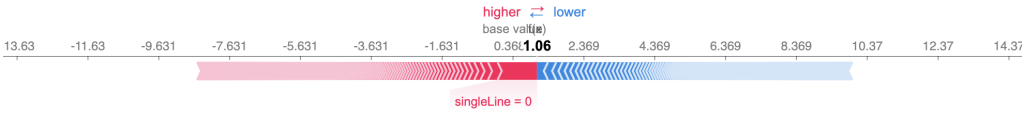
Figure 4.8: Feature Interaction.



(a) Patch for Closure-57.



(b) Patch for Math-85.



(c) Patch for Math-56.

Figure 4.9: SHAP Analysis on Patches.

while Figure 4.8(c) cannot because not enough of the test data are reaching both the two feature nodes in the tree-based boosting classifier. In the same direction, we cannot present the SHAP interaction between *singleLine* and *codeMove*. Overall, Figure 4.8 provides evidence for the impact of the interaction between learned and engineered features on the model prediction. In contrast, merging classifiers through *Ensemble Learning* does not allow for features interaction and thus fails to identify patches that were not identified using one feature set. This motivates model trainers to combine different types of features through tree-based classifiers or deep neural networks to obtain efficient deep information for identifying previously-unidentified correct patches.

Finally, Figure 4.9 presents the SHAP analyses of three patches that are exclusively

identified by classifiers built based either on learned feature set (a), or on engineered feature set (b), or on combined feature set (c). We note that contributions of each learned feature is small and it is the sum of contributions that lead to a prediction. In contrast, contributions of engineered features are significantly larger for several features. When the sets are combined, engineered features are contributing in the top, their contributions are impactful, while learned features still contribute, each, to a lesser extent. Overall, few engineered features make most of the contributions for good prediction which unsurprisingly imply that the quality and relevance of engineered features are more important than the number of features.

✍ **RQ-3** ▶ *Thanks to SHAP explanations, we were able to confirm that combining engineered and learned feature sets creates interactions that impact the prediction of classifiers, leading to improved precision and recall in correctness prediction.* ◀

4.5 Experimental Insights

Refutation of literature assumption that "patches with fewer changes are more likely to be correct". In RQ-2, we leveraged similarity between buggy code and patched code to filter out incorrect patches. The hypothesis is the more similar they are, the more likely to be correct the patch is. The best performance appears in QuixBugs which only contain bugs on one single line. However, regarding Bears, Bugs.jar and Defects4j, while a large number of incorrect patches are filtered out (cf. -Recall in Table 3.6), correct patches are recalled in low numbers (cf. +Recall in Table 3.6). Or, -Recall is low while keeping high +Recall. In the RQ-5, we use ground-truth labeled developer's patches and generated patches with balanced numbers for Defects4j to avoid bias. We use SHAP to interpret the impact of features and find the most important feature is "singleLine". The feature analysis suggests that patches consisting of a single line of code (i.e., fewer changes) are more prone to being incorrect. In contrast, correct code generally necessitates changes spanning multiple lines. For example, a value of 0 for the "singleLine" feature (indicating a change across more than one line) is the most significant contributor to identifying a correct patch for Math-56, as shown in Figure 4.9(c). This insight refutes the validity of the widely spread hypothesis.

4.6 Conclusion

In this paper, we implemented a patch correctness predicting framework, Leopard, to investigate the discriminative power of the deep learned features by training machine learning classifiers to predict correct Patches. Decision Trees, Logistic Regression, Naïve Bayes, Random Forest, XGBoost, and DNN are tried with code embeddings from BERT, Doc2Vec and CC2Vec. With BERT embeddings, Leopard (with XGBoost) yielded very promising performance on patch correctness prediction with metrics like Recall at 82.1% and F-Measure at 76.5%, Leopard (with DNN) achieved the highest score with the metric Precision at 0.744 on a labeled deduplicated dataset of 2,147 patches. We further showed that the performance of these models on learned embedding features is promising when comparing against the state of the art (PATCH-SIM [2]), which uses dynamic execution traces. We further implemented Panther (an upgraded version of Leopard) to explore the combination of the learning embeddings and the engineered features to improve the performance of identifying patch correctness with more accurate classification. In conclusion, through the utilization of SHAP, we conducted an analysis to elucidate the underlying factors and classifiers responsible for patch correctness predictions. Our analysis challenges the widely spread belief that bug fixes typically involve minimal changes. Furthermore, the explanation analysis suggests that a select set of high-quality engineered features outperforms a larger set of noisy ones, encouraging researchers to focus on the development of such features. Given our approach's capability for rapid patch correctness prediction, future research should explore its integration with APR tools to more efficiently traverse large patch spaces.

Availability. All artifacts of this study are available in the following public repository:

`https://github.com/HaoyeTianCoder/Panther`

5 Predicting Patch Correctness Based on the Similarity of Failing Test Cases

In this chapter, towards predicting patch correctness in APR, we propose a novel yet simple hypothesis on how the link between the patch behaviour and failing test specifications can be drawn: similar failing test cases should require similar patches. *We then propose BATS, an unsupervised learning-based approach to predict patch correctness by checking patch* **B***ehaviour* **A***gainst failing* **T***est* **S***pecification. Experimentally, our approach outperforms state-of-the-art techniques for identifying correct patches without the need for large labeled patch datasets; as is the case with machine learning-based approaches.*

This chapter is based on the work published in the following research paper:

- **Haoye Tian**, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, no. 4 (2022): 1-30.

Contents

5.1 Overview

Patch overfitting [25, 26] is now acknowledged as one of the major blockers to the adoption of automated program repair (APR) by software practitioners. It refers to the fact that APR-generated patches often overfit to the repair (incomplete) test suite without necessarily generalizing to other test cases. In short, overfitting patches do not implement the desired behavior that the program developers would expect. Consequently, when a generated patch is validated as passing all test cases in the test suite, it is referred to as a ***plausible*** patch. Its correctness, indeed, must still be decided manually by developers. Given that existing APR approaches generate a large number of plausible patches, most of which are actually incorrect, there is a need to develop automated approaches that can filter out incorrect patches or that can rank the correct ones higher to alleviate the burden of manual assessment.

Recently, the literature has proposed various heuristics to predict patch correctness. Csuvik *et al.* [125] translate some empirical observations into a simple assumption for ranking valid patches: correct patches apply fewer changes than incorrect ones. Xiong *et al.* [2] build on the hypothesis that test case dynamic execution behaviours are different between correct and incorrect patches. Other researchers propose to focus on learning, with static features of patches, to filter out incorrect patches. For example, Ye *et al.* [27] proposed such a supervised learning-based approach after investing in careful engineering of patch features. In contrast, Tian *et al.* [3] relied on deep representation learning of code changes for yielding patch embeddings that are fed to a supervised learning system.

Overall, existing research in patch correctness prediction has provided promising performance [33]. Nevertheless, they suffer from various caveats. On the one hand, the state of the art dynamic-based approaches require the expensive execution of test cases, which unfavorably impacts the efficiency of the patch assessment process. Besides, such approaches are challenged in practice by the oracle problem: given a test case, we do not always have an accurate specification of what the output should be [185]. On the other hand, static-based approaches often require a substantial analysis effort to identify adequate features and properties. In addition, machine learning-based approaches require many labeled patch samples (both correct and overfitting patches). They further exhibit issues of generalization beyond the projects they have been trained on [33].

In their seminal study on patch plausibility and correctness, Qi *et al.* [25] have presented Kali, a system that performs "repair" by only removing or skipping code in programs. Kali generated several patches that pass many weak test suites. Our postulate, however, is that Kali-generated patches should be readily-identifiable as *plausible but incorrect* in an APR pipeline: it is unlikely that fixing a program that presents a bug in array iteration would require simply removing whole statements. This calls for research to assess the behaviour of the patch (i.e., what it does) against the nature of the bug (i.e., as expressed by the failing test case specification).

The idea of checking patch behaviour against failing test specification has not yet been fully exploited in the literature. Recent work by Ye *et al.* [27] and Tian *et al.* [3] do not even reason about the test cases. The approach by Xiong *et al.* [2] builds on heuristics that consider the similarity of execution behaviours of passing test cases on original and patched programs. Their work, however, does not try to answer the specific question of ***whether the generated correct patch is actually relevant***

to the failing test case(s) triggering the repair process. This is the key hypothesis we introduce and validate:

> **"When different programs fail to pass similar test cases, it is likely that these programs require similar code changes."**

Similarity thus becomes a key challenge: syntactic similarity is not sufficient as it would restrict the search to type-1 or type-2 clones. We have to explore similarity measurements that can capture semantic relationships. Recent work [167, 163, 14, 186, 187] in learning distributed representations of code and code changes have been shown to preserve some semantics (beyond token similarity) and have yielded promising models that were effective on a variety of downstream tasks.

This paper. We build on the aforementioned hypothesis to investigate the possibility of predicting patch correctness by clustering test cases and patches. In this paper, we rely on recent approaches for code representation learning to reason about code and patch similarity.

❶ [**Heuristic**] We propose a novel heuristic on the relationship between patches and their failing test cases. Although the intuition behind this heuristic is basic and hinted at in regression testing literature [188], we are the first to propose and validate it in the APR patch assessment area.

❷ [**Validation**] We present a comprehensive validation of the hypothesis that similar test cases are associated with similar patches. Concretely, we consider the case of developer-written patches in the Defects4J dataset and leverage various distance metrics to perform hierarchical clustering based on the embeddings of test cases and patches.

❸ [**BATS**] Based on the heuristic, we propose BATS (Behaviour Against failing Test Specification), an approach to predict patch correctness by statically checking the similarity of generated patches against past correct patches that correspond to failing test cases which are similar to the failing tests of the bug under resolution. More specifically, given one buggy program with its failing test cases and the APR-generated patches, BATS first enumerates similar failing test cases within the search space of historical bugs. Then, BATS calculates the similarity between the correct patches associated with the identified failing test cases and the APR-generated patches. Finally, APR-generated patches with similarity scores above a predefined threshold t are predicted as correct while patches with similarity scores lower than t are predicted as incorrect. The artifact of this study is publicly available at **`https://github.com/HaoyeTianCoder/BATS`**.

❹ [**Evaluation**] After collecting a large dataset of plausible patches generated by 32 APR tools or extracted from defects benchmarks, we apply BATS and measure its performance in classifying correct patches.
- Overall, BATS achieves an AUC (Area Under Curve) between ∼0.56 and ∼0.72 and a recall between ∼56% ∼84% in identifying correct patches.
- When comparing with a recent supervised learning classifier [3], BATS improves the recall in identifying correct patches by 7 percentage points and achieves an equivalent recall in excluding incorrect patches.
- Comparing against the state-of-the-art dynamic approach PATCH-SIM [2], BATS outperforms it with higher scores of AUC, F1 +Recall and -Recall

for the subset of patches where BATS can find similar test cases.

– We note that the performance of BATS is impacted by the search space for finding similar test cases. Therefore, after demonstrating the promise of the proposed heuristic, we show that it is worthwhile to combine BATS with the state-of-the-art approaches in order to improve performance in identifying correct patches. To that end we consider a usage scenario where similar test cases are lacking to apply BATS. Instead, we investigate whether BATS can still be used as a supplement to another approach: when BATS is integrated with the recent supervised learning classifier [3] , the overall recall in detecting correct patches is improved with 5 percentage points. The experimental results also show that BATS can be complementary to PATCH-SIM [2] to recall more correct and exclude more incorrect patches.

5.2 Approach

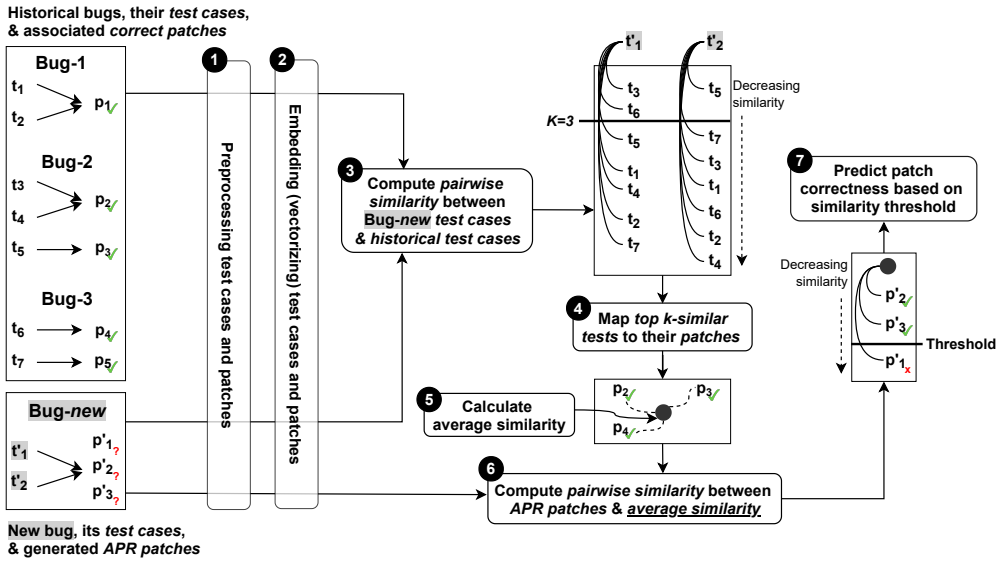


Figure 5.1: Overview of BATS.

In this section, we present BATS, our approach to predict patch correctness based on the similarity of the new patch to known correct patches with *similar failing test cases*. Figure 5.1 gives an overview of the approach. BATS assumes the following inputs: (i) The bug under resolution, its *associated failing test cases*, and *APR-generated plausible patches*, and (ii) Historical bugs with *their failing test cases* and the *associated known correct patches*.

To predict patch correctness, BATS performs the following steps: (1) Pre-process patches and test cases to prepare them for the embedding step, (2) Embed patches and test cases into higher dimensional space, (3) Compute pairwise similarity between *the failing test cases of the bug under resolution* and the *historical test cases*, (4) Select at most top-k historical test cases with similarity score greater than t_{Test}, and map them to their associated correct patches, (5) Compute an average similarity among selected historical known correct patches, (6) Compute pairwise similarity between every plausible patch for the bug under resolution and the average similarity of known correct patches from the previous step, and (7) Predict a plausible patch as correct if its similarity to the average similarity of correct patches (from step 5) is greater than some threshold t_{Patch}, otherwise, BATS predicts the patch to be incorrect.

5.2.1 Pre-processing Test Cases and Patches

BATS leverages the similarity of test cases and their corresponding patches to predict patch correctness. Therefore, the first two steps of BATS aim at preparing the test cases and patches into a form suitable for computing similarity. One way to achieve this is by embedding patches and test cases into higher dimensional space to obtain numerical vectors that are suitable for vector similarity computations.

Tokenizing test cases BATS treats the source code of individual test methods as sequences of tokens while also using camelCase tokenization to further breakdown identifiers into their sub tokens.

Tokenizing patches Patches are tokenized in the same manner as test cases with two differences. First, BATS considers changed lines without their contexts. I.e., it

selects added and removed lines only, marked with '+' and '-', respectively. Second, to keep the information about added and removed lines, BATS keeps the '+' and '-' markers as part of each patch line.

5.2.2 Embedding Test Cases and Patches

BATS relies on similarity calculations between test cases and between patches. Therefore, the second step is to embed test cases and patches into a higher dimensional space to enable similarity computations. An embedding is a function that maps each token into a high dimension real-value vector while maintaining semantic similarities between similar tokens.

In our approach, embedding individual test methods is straightforward but it is not the case for patches. Patches are composed of several individual hunks (contiguous changes) while the order of the hunks is irrelevant to the patch. Each hunk can be embedded as a sequence of tokens into a single vector. But how can we combine the different hunks to obtain a single vector representing the entire patch? Simple concatenation of the vectors of different hunks does not produce a unique vector. Because there is no specific order for the individual hunks, different orderings of the different hunks produces different vectors for the same patch. Therefore, instead of concatenating vectors of different hunks, we sum the vectors of the different hunks to obtain a unique vector representing the entire patch.

To obtain the embeddings for test cases and patches, we leverage three state-of-the-art pre-trained models:

- **code2vec.** Alon *et al.* [167] leverage the AST representation of a method to produce its embedding. code2vec has been applied to a variety of downstream tasks, including predicting method names, where it revealed its ability to learn the structure and semantics of code fragments. We propose in this work to leverage code2vec for embedding test cases. To that end, we build on a pre-trained model provided by the authors [167] who trained the model on a dataset containing ∼14 million Java methods.
- **CC2Vec.** Hoang *et al.* [163] introduced the CC2Vec hierarchical attention neural network model for learning vector representations of patches. In the training phase, the learning is guided by the commit messages that are associated with patches and uses them as semantic descriptions of the patches. We propose in this work to leverage CC2Vec for embedding APR-generated patches. We consider the same architecture where we skip the feature crossing layer and train a new model building on the large dataset of 24,000 patches provided by the authors.
- **BERT.** Another popular embedding method that is applied to natural language is BERT [155], a transformer-based self-supervised language model. Recent work in software engineering fine-tuned BERT on code fragments and applied it to produce embeddings that were shown effective [162, 169]. Tian *et al.* [3] recently leveraged BERT in their experiments on filtering out incorrect patches based on the similarity between buggy code and patched code. For comprehensive experiments, we also propose to investigate using BERT for embedding patches.

5.2.3 Finding Similar Test Cases

The major hypothesis of BATS is that similar failing test cases have similar associated patches. Therefore, the third step of the approach is to find similar test cases from historical fixed bugs, i.e., test cases that failed before their associated

fixes are applied, and are similar to the failing test cases of the current bug under resolution. To this end, BATS computes pairwise similarities between each failing test of the bug under resolution and all tests found in the search space of BATS. To compute the similarity between test cases, we apply the Euclidean distance to their code2vec embeddings. Then using a similarity threshold, t_{Test}, BATS selects at most the top-k historical tests with similarity score $> t_{Test}$. If the number of top k similar test cases is smaller than the number of tests with similarity score $> t_{Test}$, k is adjusted accordingly. Note that historical failing test cases do not need to be from the history of the same project of the bug under resolution.

5.2.4 Mapping Historical Failing Test Cases to their Patches

When BATS finds the most similar test cases that failed in the past, BATS further maps these historical failing tests to their associated correct patches which were applied and accepted as correct fixes for those failing tests. Our hypothesis is that a plausible patch for the current bug under resolution is correct if it is similar to these historical correct patches because their failing test cases are also similar. To facilitate the comparison of the plausible patches to these historical correct patches[1], BATS averages the historical correct patches by computing an *average* of their embedding vectors.

5.2.5 Predicting Patch Correctness

To predict whether a given plausible patch is correct or not, BATS calculates the similarity between this patch and the average of the historical correct patches obtained in the previous step. BATS computes the similarity of patches through Euclidean and Cosine similarity measurements. If this similarity score is higher than a threshold t_{Patch}, BATS predicts that this patch is correct, otherwise, the plausible patch is predicted as incorrect. In our experiments, we set $t_{Patch} = 0.5$.

5.2.6 An Example

Consider the following example of bug Chart-26 from the Defects4J dataset. Chart-26 triggers 22 test cases to fail. To fix this bug, APR tools SOFix and KaliA generate the two plausible patches presented in Figure 5.2 and Figure 5.3, respectively.

```
--- .../source/org/jfree/chart/axis/Axis.java
+++ .../source/org/jfree/chart/axis/Axis.java
@@ -1189,10 +1189,12 @@
         }
         if (plotState != null && hotspot != null) {
             ChartRenderingInfo owner = plotState.getOwner();
+            if (owner != null) {
                 EntityCollection entities = owner.getEntityCollection();
                 if (entities != null) {
                     entities.add(new AxisLabelEntity(this, hotspot,
                             this.labelToolTip, this.labelURL));
                 }
+            }
         }
         return state;
```

Figure 5.2: A correct patch generated by APR SOFix for the Defects4J bug Chart-26.

[1]The patches were written by developers to fix the related bugs and were committed in the historical repository of related projects.

```
--- .../source/org/jfree/chart/plot/CategoryPlot.java
+++ .../source/org/jfree/chart/plot/CategoryPlot.java
@@ -2541,7 +2541,9 @@

         // record the plot area...
         if (state == null) {
         // if the incoming state is null, no information will be passed
+        if (true)
+             return;
         // back to the caller - but we create a temporary state to record
         // the plot area, since that is used later by the axes
         state = new PlotRenderingInfo(null);
```

Figure 5.3: An incorrect patch generated by APR KaliA for the Defects4J bug Chart-26.

First, to find out whether any of the two patches is correct, BATS looks for test cases that are similar to the 22 failing test cases in the available search space. The search space includes the history of the Chart project as well as other projects, when available. Second, BATS, thanks to its code2vec-based similarity checker, identifies two test cases as similar to some of the 22 failing tests: one test case is associated with bug Chart-4 and the other is associated with bug Chart-25. A manual investigation of the semantics of these two test cases reveals that they indeed aim at detecting unhandled Null pointer dereferences.

Third, after BATS maps the two identified historical test cases to their corresponding correct patches, it measures the similarity of the APR-generated patches to these relevant correct patches. Finally, based on the similarity score and a similarity threshold, BATS precisely predicts that the generated patches by SOFix and KaliA are correct and incorrect, respectively. When we manually inspect the historical correct patches that were applied to fix Chart-4 (cf. Figure 5.4) and Chart-25 (cf. Figure 5.5), we notice that they both implement similar behavior (i.e., adding null check) as the proposed patch by SOFix (cf. Figure5.2). On the contrary, the KaliA patch (cf. Figure 5.3) suggests an irrelevant code change.

```
--- .../source/org/jfree/chart/plot/XYPlot.java
+++ .../source/org/jfree/chart/plot/XYPlot.java
@@ -4490,6 +4490,7 @@ public class XYPlot extends Plot implements ValueAxisPlot,
     Pannable,
                     }
                 }

+                if (r != null) {
                     Collection c = r.getAnnotations();
                     Iterator i = c.iterator();
                     while (i.hasNext()) {
@@ -4498,6 +4499,7 @@ public class XYPlot extends Plot implements ValueAxisPlot,
     Pannable,
                             includedAnnotations.add(a);
                         }
                     }
+                }
             }
         }
```

Figure 5.4: A correct developer-written patch for the Defects4J bug Chart-4.

In our evaluation, there are 17 plausible patches generated by different APR tools for Chart-26[2]. In Figure 5.6, we see the 17 patches ranked according to their similarity score computed by BATS. On the x-axis, each patch is labeled with a combination of the name of the APR tool that generated it and a numerical id as a

[2]These APR-generated patches have been labeled in previous work.

```
--- .../source/org/jfree/chart/renderer/category/StatisticalBarRenderer.java
+++ .../source/org/jfree/chart/renderer/category/StatisticalBarRenderer.java
@@ -256,6 +256,9 @@ public class StatisticalBarRenderer extends BarRenderer

         // BAR X
         Number meanValue = dataset.getMeanValue(row, column);
+        if (meanValue == null) {
+            return;
+        }

         double value = meanValue.doubleValue();
         double base = 0.0;
@@ -312,7 +315,9 @@ public class StatisticalBarRenderer extends BarRenderer
         }

         // standard deviation lines
-            double valueDelta = dataset.getStdDevValue(row, column).doubleValue
    ();
+        Number n = dataset.getStdDevValue(row, column);
+        if (n != null) {
+            double valueDelta = n.doubleValue();
             double highVal = rangeAxis.valueToJava2D(meanValue.doubleValue()
                     + valueDelta, dataArea, yAxisLocation);
             double lowVal = rangeAxis.valueToJava2D(meanValue.doubleValue()
@@ -341,6 +346,7 @@ public class StatisticalBarRenderer extends BarRenderer
             line = new Line2D.Double(lowVal, rectY + rectHeight * 0.25,
                                      lowVal, rectY + rectHeight * 0.75);
             g2.draw(line);
+        }

         CategoryItemLabelGenerator generator = getItemLabelGenerator(row,
                 column);
```

Figure 5.5: A correct developer-written patch for the Defects4J bug Chart-25.

tool may generate more than one patch. We note that most of the correct patches (grey bars) are ranked ahead of incorrect patches (white bars) which confirms that our hypothesis is effective in discriminating correct patches from incorrect ones.

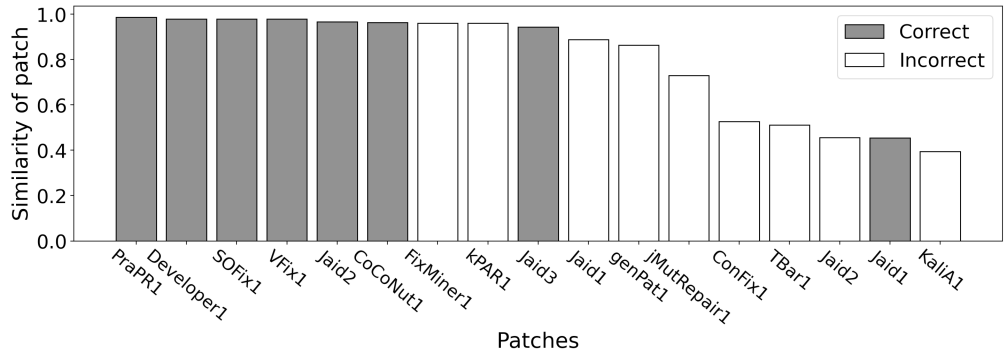


Figure 5.6: The ranked patches generated by APR tools for Chart-26. The numerical value next to each tool name indicates patch id since a tool can generate more than one patch.

5.3 Study Design

In this section, we introduce the experimental setup to evaluate our hypothesis and our approach, BATS. We present the research questions in Section 5.3.1, the datasets in Section 5.3.2, and the evaluation metrics in Sections 5.3.3 and 5.3.4.

5.3.1 Research Questions

Our research questions aim to validate the hypothesis, draw insights for developing a prediction method for patch correctness and finally assess the BATS approach with APR-generated plausible patches while comparing against recent state of the art approaches.

RQ-1. *Does the similarity of bug-triggering test cases correlate with the similarity of the associated bug fixing patches?* This research question aims to validate the feasibility of our proposed hypothesis. To this end, we conduct two experiments: clustering similar test cases and assessing patch similarity with the similar test cases. This offers insights into the design of the patch correctness identification system based on inferred thresholds.

RQ-2. *To what extent can an approach assessing patch behaviour against test specification based on unsupervised learning be effective in identifying correct patches among plausible ones?* With this research question, we first present the implementation of BATS and evaluate its performance in identifying correct patches in 1,278 patches generated by 32 APR tools.

RQ-3. *Can BATS achieve competitive results against the recent state of the art approaches?* In this research question, we compare BATS against static and dynamic approaches of predicting patch correctness for APR tools. Then we explore the possibility of leveraging BATS to complement the state of the art in identifying correct patches.

5.3.2 Datasets

We focus our experiments on the Defects4J [88] benchmark since it is widely used in the literature, and we can readily collect plausible patches generated by APR tools on the programs included in the benchmark. Table 5.1 provides the statistics on the collected patches. Besides the 205 developer (correct) patches provided in the benchmark, we also leverage the reproduced dataset from the study of 16 APR systems by Liu et al. [175], which we augment with a dataset provided by Ye et al. [182]. Finally, we also scan the artifacts released in the literature towards identifying plausible patches generated by recent APR tools. Overall, we share with the community the largest dataset[3], to-date, of patches for Defects4J bugs, which includes hundreds of overfitting (i.e., incorrect) patches.

Dataset per experiment: For answering RQ-1, we rely on the 1,120 failing test cases and the associated 205 developer patches in the Defects4J dataset. For answering RQ-2 and RQ-3 (assessment of the BATS approach), we consider the 1,278 plausible patches generated by APR tools or provided by the Defects4j project developers. In these RQs, we also rely on the 1,120 test cases and 205 developer patches as the search space for similar test cases to the failing test case being addressed. We ensure, however, that for every execution of BATS we remove from

[3] `https://github.com/HaoyeTianCoder/BATS`

Table 5.1: Statistics on the dataset of developer-written and APR-generated patches.

Subject	Patches	Incorrect	Correct	Subject	Patches	Incorrect	Correct
Developer [88]	205	0	205	**jGenProg2015** [189]	11	8	3
3sFix [190]	60	57	3	**jKali** [191]	14	12	2
ACS [44]	21	6	15	**jMutRepair** [191]	15	12	3
ARJA [192]	183	168	15	**KaliA** [192]	14	14	0
CapGen [46]	64	39	25	**kPAR** [100]	50	42	8
Cardumen [193]	10	8	2	**LSRepair** [147]	18	15	3
CoCoNut [194]	28	0	28	**Nopol2015** [189]	10	8	2
ConFix [195]	66	52	14	**PraPR** [113]	22	0	22
DeepRepair [196]	13	9	4	**RSRepiarA** [192]	18	18	0
DynaMoth [59]	18	18	0	**SequenceR** [65]	35	24	11
ELIXIR [197]	36	13	23	**SimFix** [132]	35	12	23
FixMiner [19]	27	17	10	**SketchFix** [130]	21	9	12
GenPat [45]	29	18	11	**SOFix** [17]	21	2	19
GenProgA [192]	14	14	0	**ssFix** [129]	19	8	11
HDRepair [16]	6	2	4	**TBar** [50]	57	36	21
Hercules [146]	51	14	37	**VFix** [102]	23	1	22
JAID [108]	64	33	31				
				All	1,278	689	589

the search space the failing test cases and the developer patches that are related to the bug under resolution.

Figure 5.7 presents the distribution of 1,278 generated patches for Chart, Lang, Math, and Time projects that have been widely used in the community of automated program repair [175, 29] and patch correctness identification [2, 3].

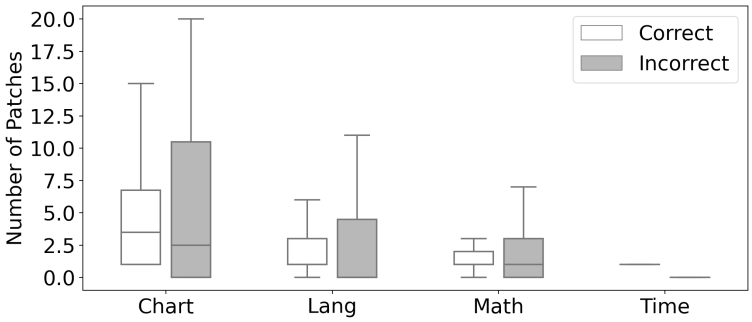


Figure 5.7: Distribution of the number of collected patches per project in the Defects4j dataset.

5.3.3 Cluster Analysis Metrics

To group similar test cases and similar patches together respectively, we explore unsupervised learning algorithms to perform clustering. K-means [198] generally provides a good clustering performance and strong interpretability. Its main limitation, however, is that it requires the user to specify the number of clusters k, which is unfortunately specific to the datasets. We therefore adopt a hierarchical clustering algorithm, e.g., bisecting K-means [199], to empirically determine the appropriate value of k.

Bisecting K-means. This algorithm was initially proposed to overcome the challenges of K-means (local minima, non-spherical clusters, etc.). Bisecting K-means modifies the K-Means algorithm to produce partitional/hierarchical clustering, thus recognizes clusters of any shape and size. The algorithm starts by splitting the dataset into two clusters based on K-means. Then, iteratively, each cluster is split. Each time the two clusters are identified as those presenting the smaller sum of squared errors (SSE). By placing a threshold for the performance in terms of the sum of squared errors, the clustering iterations can be stopped. This algorithm is therefore convenient to infer a reasonable number k of clusters in a dataset.

Cluster Similarity Coefficient (Adjusted Silhouette Coefficient). Once clusters are yielded, we can measure to what extent each element is actually similar to its own cluster (cohesion) compared to other clusters (separation). To that end, we propose a similarity coefficient (SC) and cluster similarity coefficient (CSC) metrics of each cluster element based on its internal similarity and its external similarity towards other elements. We use the following equations:

$$SC(e) = \frac{in(e) - out(e)}{\max\{in(e), out(e)\}} \quad (5.1) \qquad CSC = \frac{1}{n}\sum_{e=1}^{n} SC(e) \quad (5.2)$$

where $in(e)$ represents the average Euclidean Similarity from the (test case or patch) embedding e to other embeddings in the same cluster; $out(e)$ represents the average Euclidean Similarity from the embedding e to the embeddings in other clusters. The value range of SC is [-1,1]: the larger the value of SC, the better the clustering effect. When SC value is greater than 0, the clustering is consistent. And in order to measure the overall performance, CSC (averaging SC) is used to calculate the coefficient taking into account all clusters.

Sum of Squared Error (SSE). Finally, we rely on the commonly-used sum of squared errors to measure the variance within clusters. SSE in our study is computed as the sum of the squared differences between each embedding and its cluster's mean. If within each and every cluster, all cases are identical, the SSE would then be equal to 0 as per equation 5.3.

$$SSE = \sum_{i=1}^{k}\sum_{j=1}^{n_i}(x_{ij} - x_i)^2 \quad (5.3)$$

where k represents the number of clusters, n_i represents the number of elements of the i-th cluster, x_{ij} represents the j-th element of the i-th cluster, and x_i represents the center element of the i-th cluster. Therefore, the smaller the SSE, the better the clustering effect.

5.3.4 Performance Metrics

We consider the **Recall** of BATS in two dimensions:

- **+Recall** measures to what extent correct patches are identified, i.e., the percentage of correct patches that are indeed predicted as correct [3].
- **-Recall** measures to what extent incorrect patches are filtered out, i.e., the percentage of incorrect patches that are indeed predicted as incorrect [3].

$$+Recall = \frac{TP}{TP + FN} \quad (5.4) \qquad -Recall = \frac{TN}{TN + FP} \quad (5.5)$$

where TP represents true positive, FN represents false negative, FP represents false positive, TN represents true negative.

Area Under Curve (AUC) and F1. By considering the similarity score as a prediction probability, BATS can be evaluated like any machine learning model with the common metrics such as AUC (the overall ability to distinguish between correct and incorrect patches) and F1 score (harmonic mean between precision and recall for identifying correct patches).

MAP and MRR. Since BATS ranks all generated patches based on similarity scores, instead of simply considering the top-1 as the correct, we can consider a recommendation and leverage common metrics used in assessing the ranked list. The

mean average precision (MAP) and mean reciprocal rank (MRR) are such metrics that help assess whether BATS can place correct patches ahead of incorrect patches in the ranked list presented to the developers.

$$MAP = \frac{1}{n}\sum_{i=1}^{n}\frac{\sum_{j=1}^{m}(P_{ij} \times Rel_{ij})}{\#\text{ correct patches}} \quad (5.6) \qquad MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i} \quad (5.7)$$

Where Rel_{ij} is an indicator function that is 1 if the j-th patch is correct in list i, otherwise it is 0. P_{ij} is the precision at the threshold j in the list i, and the $rank$ represents the first correct ranking.

BATS relies on the unsupervised learning technique and considers to set thresholds of similarities among test cases and patches to predict the correctness of patches. Given that in practice we cannot tune the decision threshold based on the specific case of each bug, we fix a single similarity threshold to compute the accuracy, precision, and false positives. We set the model prediction threshold to 0.5. As soon as the normalized similarity score between the generated patch and the identified cluster of historical patches is higher than 0.5, we predict the patch as a correct one. Otherwise, we predict it as incorrect. We note, however, that in the literature, some approaches [2] are assessed by adjusting the threshold in the test data to achieve the best possible +Recall.

5.4 Experiments and Results

In this section, we first present the validation of the hypothesis in our work (Answer to RQ-1 in Section 5.4.1). Then , we report the experimental results of patch correctness prediction on our collected dataset (Answer to RQ-2 in Section 5.4.2 and RQ-3 in Section 5.4.3).

5.4.1 RQ-1: Cluster of Similar Test Cases and Patches

[Objective]: We perform experiments to answer RQ-1, whether the proposed hypothesis is valid for patch correctness identification with the following two sub questions to observe whether failing test cases are similar when the associated patches are similar. First, we investigate whether the 1,120 failing test cases in Defects4J can be grouped in clearly separable clusters. Based on each such test cluster, we decide that the associated 205 developer patches constitute a cluster. Then, we seek to validate the feasibility of leveraging similar test cases to predict the patch correctness with Defects4J dataset.

- **RQ-1.1** *Do patches cluster well together when their test cases are similar?* To answer this RQ, we automatically cluster test cases into groups of similar test cases, then we assess whether the associated patches in each group also have a good clustering cohesion.
- **RQ-1.2** *Given two test cases, can their similarity score be used as an indicator of their relatedness in terms of patch similarity?* We investigate test case similarity vs. patch similarity hypothesis in a fine-grained manner beyond the clusters.

[Experimental Design for RQ-1.1]: First, we use the code2vec and CC2Vec pre-trained models to produce embeddings for each test case and for each patch respectively in the Defects4J dataset. To cluster test cases, we rely on the bisecting K-means algorithm to produce hierarchical clusters. At each iteration of bisecting K-means, we compute the sum of squared error of the clusters. We use the evolution of reduction in SSE values to decide on a threshold for the number of clusters into which we can split the test cases in our dataset. Experimentally, we observed that the number of clusters k for which the SSE saturates, i.e., no longer drops, is 40. Additionally, we empirically validate the consistency of our proposed hypothesis by investigating the Cluster Similarity Coefficient (CSC) of different cluster settings (i.e., $k \in \{30, 40, 50\}$).

In this experiment, we leverage the Similarity Coefficient (SC) and Cluster Similarity Coefficient (CSC) (cf. Section 5.3.3) to assess the consistency and cohesion of the yielded clusters. Nevertheless, we can observe the clustering effect for test cases by investigating the distance distribution of each test case to the center of its cluster: we first consider the distance[4] with all test cases in a single group. Then, we compute the distance of each test case to the center of its K-means-inferred cluster. We consider that if the distribution of distances shows that, on average, distances in a cluster are lower than in the whole group, then the test cases have been well-separated. We confirm that when considering the whole dataset, the distances are the longest (i.e., the median value of distance distributions for all clusters are lower than the one for the whole dataset). In several inferred clusters, the median

[4]Distance and similarity are two concepts that are used interchangeably in this section depending on the context to facilitate comprehension.

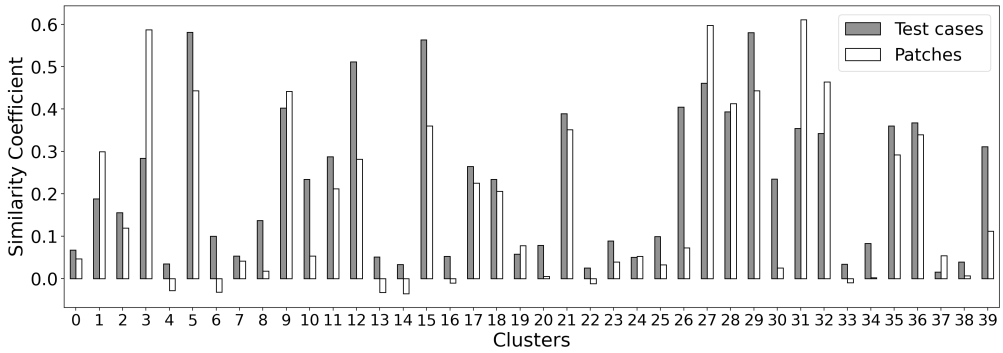


Figure 5.8: Similarity coefficient of test cases and patches at each cluster.

distance (of the test case to the cluster center) is halved. These observations suggest indeed that the test cases could be readily clustered.

[Experimental Results for RQ-1.1]: Considering the test cases clusters, we infer associated groupings of patches that address these test cases. We then compute the CSC to evaluate the consistency and cohesion of the patch clusters that we have derived. These metrics evaluate the distances among patches within each cluster and the distances among clusters. We validate the metrics on the overall Defects4J ground-truth dataset.

Table 5.2 presents the cohesion and consistency metrics for the patches regrouped based on the clusters of test cases when the number of clusters is 30, 40 and 50, respectively. The Cluster Similarity Coefficient(CSC) is the average value of similarity coefficient (SC) values for all clusters. "**Qualified**" represents the ratio of clusters that have $SC > 0$ out of all clusters identified. We compare those metrics (on test case clusters) to the associated patch clusters. The positive values of the CSC indicate that, on average, the elements inside the same group are indeed more similar among themselves than they are similar to the elements in other groups. When test cases are well grouped ($CSC > 0$), the corresponding clustering of the associated patches clusters together consistently ($CSC > 0$). In more details, a large ratio of clusters (33/40) also have high cohesion. When adjusting the number of clusters to 30 or 50, the results show that the major clusters of patches (23/30, 37/50) still keep high cohesion, and the clustering of test cases and patches are consistent to each other ($CSC > 0$).

Table 5.2: Statistics on the performance of clustering of test cases and patches with 30, 40 and 50 clusters.

Subjects	Cluster Similarity Coefficient	Qualified
Test Cases	0.19	30/30
Patches	0.16	23/30
Test Cases	0.19	40/40
Patches	0.16	33/40
Test Cases	0.21	50/50
Patches	0.14	37/50

Figure 5.8 further presents the similarity coefficient (SC) of test cases and patches for each cluster when k is set to 40. We observe that, for most pairwise clusters of test cases and patches, when the SC value of test cases (presented with grey bar) in one cluster is high, the associated patches (presented with white bar) also have a high SC score. We further calculate a Pearson correlation between the clusters of test cases and the clusters of associated patches, of which value is $0.883 > 0$. Pearson

correlation can be used to measure the linear correlation between two sets of data where a higher positive value (i.e., > 0) indicates a more positive association. Such results indicate that the similar test cases can lead to similar patches.

> **[RQ-1.1]** *Given a cluster of similar test cases, their associated patches cluster consistently. This experiment also hints that the representation models for test cases and clusters yield meaningful embeddings for investigating the relatedness of patches and test cases.*

[Experimental Design for RQ-1.2]: The clustering experiment for RQ-1.1 focuses on average distances within clusters, we further seek to validate the possibility of using test case similarity as a potential heuristic to predict correct patch behavior. The objective is to answer "*whether the similarity of two test cases can be used as an indicator of their relatedness in terms of patch similarity*". To this end, we first consider finding the most similar test case from the search space of historical test cases for the failing-executed test case of a given bug. We then assess to what extent the patch of a given bug is similar to the patch associated to the most similar test case (referred to ① **Scenario H**), of which results are compared against the results in ② **Scenario N** where we compute the average similarity between a given patch and all other patches. The experiments finally investigate scenarios where the closest test case is sought within the all project or only in other projects (excluding the one where the test case is found).

[Experimental Results for RQ-1.2]: Figure 5.9 presents the overall similarity distribution between each of the 1,120 test cases in the dataset and its closest counterpart: while some test cases indeed have very similar counterparts, many test cases have low similarities with their closest counterparts. These relatively low similarities for many test cases can be explained by the limited number of test cases considered in the study datasets. These results suggest that it may not always make sense, for a given test case, to blindly consider the most similar counterpart since this counterpart can still be highly dissimilar. We thus propose to experimentally determine a threshold to decide when in practice the closest test should not be considered as similar.

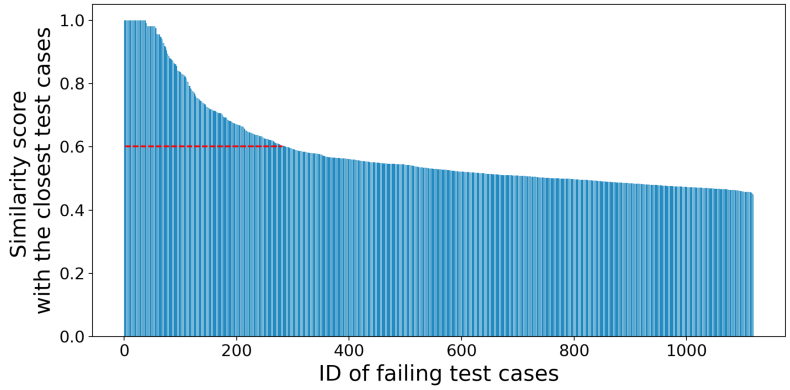


Figure 5.9: Distribution on the similarities between each failing test case of each bug and its closest similar test case.

In Figure 5.10, we compare the distributions of the similarity scores for the two scenarios H and N. In all projects, the distributions in the scenario H of our hypothesis present higher similarities. This indicates that the most similar test case

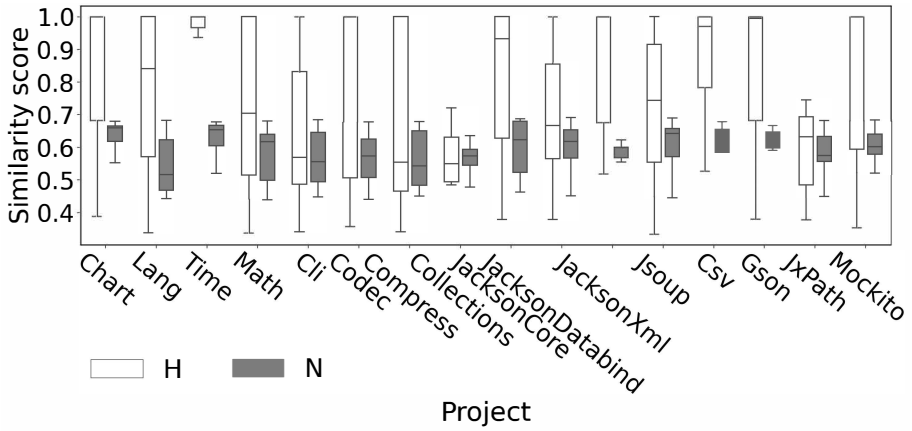


Figure 5.10: Distributions on the similarities of pairwise patches (similar patch selected with Scenario H vs. Scenario N from **all projects**, i.e., the search space for searching similar cases is all projects in the dataset).

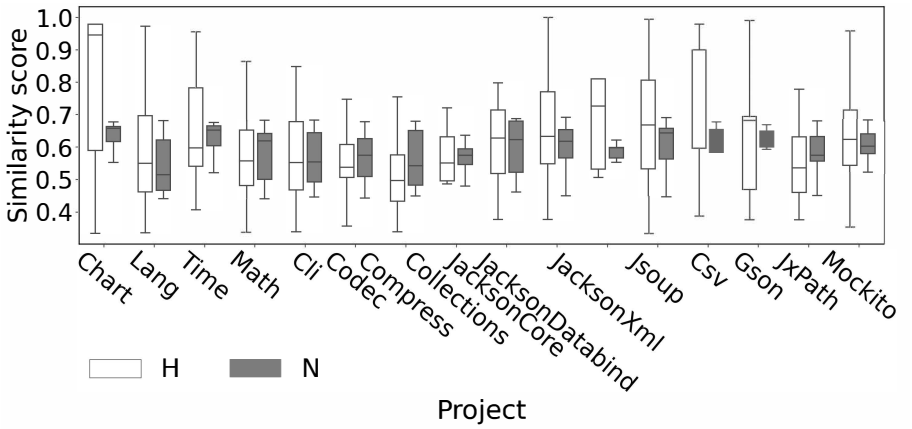


Figure 5.11: Distributions on the similarities of pairwise patches (similar patch selected with Scenario H vs. Scenario N from **other projects**, i.e., the search space for searching similar cases does not include the buggy project itself).

is a good proxy to identify a patch[5] that will be more similar than the average patch in a dataset.

In the aforementioned experiment, the test cases in search space are allowed from the same project due to the lack of test cases. To evaluate our hypothesis in the scene of insufficient test cases, we further reproduce the comparison by focusing on test cases and patches that are from other projects (i.e., the buggy project itself is excluded from the search space of test cases). Figure 5.11 further provides the distributions for the two scenarios H and N. In this case, we note that the difference is less pronounced for most projects. We postulate that this is due to the fact that similarity scores are low. Thus we propose to set a threshold and consider, for the scenario H, cases where the test cases present a higher similarity than the threshold.

When considering each of the 1,120 test cases, for many of them the closest test case in the search space is actually not a "similar" one. By looking at the pairwise similarity distribution shown in Figure 5.9, we note that 74% (831/1120) similarity values are lower than 0.6. We therefore arbitrarily decided to use this value as the threshold[6] to explore the hypothesis by isolating the minority of cases where the similarities are significant (more experiments with different thresholds are presented in Section 5.4.2 for RQ-2). Figure 5.12 presents the comparison of

[5]This patch is the one associated with the similar test case that failed in the past.

[6]Note that it aims to validate our hypothesis but not to infer a specific/adaptable threshold.

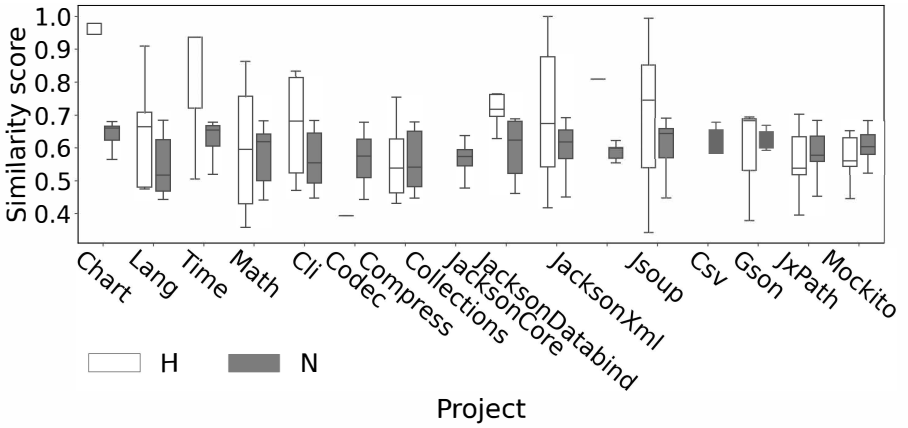


Figure 5.12: Distributions on the similarities of pairwise patches (similar patch selected with Scenario H vs. Scenario N from **other projects**, i.e., the search space for searching similar cases does not include the buggy project itself, by setting the threshold at 0.6).

similarity distribution when the patch pairs are selected with Scenarios H and N from other projects, after setting a threshold of 0.6 for the similarity of test cases in Scenario H to reduce noise. We observe that scenario H now provides the highest similarities for the paired patches (based on the similarities of test cases). Note that some white boxes (scenario H) in the plot are missing is due to lack of high enough similar test cases.

[**RQ-1.2**] *Given a test case and its most similar test case, their associated patch pair will exhibit a similarity that is statistically higher than the average similarity for all pairs of patches in the same project. This finding is also confirmed across projects.*

5.4.2 RQ-2: Identifying Correct Patches with BATS

[**Objective**]**:** Findings in answering the above RQ confirm our hypothesis that test case similarity correlates with patch similarity. BATS is therefore implemented to explore this hypothesis scenario in an APR setting where generated plausible patches (for a failing test case T) are ranked based on their similarity with a set of historical patches that were applied by developers (to address failing test cases similar to T). To answer this RQ, we leverage the 1,278 plausible patches which are composed of 205 developer-written patches and 1,073 APR-generated patches by the tools in Table 5.1.

By assessing BATS on the collected dataset of plausible patches generated by literature APR tools, our main aim is to demonstrate to the community the feasibility of the proposed research direction for patch assessment.

[**Overall Assessment**]**:** We first design a baseline with a simple hypothesis which considers that a patch is more likely to be correct if it is similar to some historically correct patches (i.e., any correct patches). In contrast to BATS, this baseline does not consider failing test cases as the constraint for reducing the search space. We recall that the performance of BATS, which relies on search, is dependent on the availability (in project repositories) of test cases that are actually similar to the failing test case addressed by the APR-generate patches. As introduced earlier, the closest test cases in the search space may actually not be that similar: this is a

Table 5.3: Baseline's performance on identifying (in)correct patches.

Patch embedding†	Correct	Incorrect	AUC	F1	+Recall	-Recall
CC2Vec	589	689	0.586	0.579	0.705 (415)	0.379 (261)
Bert			0.593	0.558	0.647 (381)	0.428 (295)

Table 5.4: BATS's performance on identifying (in)correct patches.

Patch embedding†	T*	# Correct patches	# Incorrect patches	AUC	F1	+Recall	-Recall
CC2Vec	0.0	589	689	0.557	0.549	0.628 (370)	0.437 (301)
	0.6	144	181	0.559	0.505	0.562 (81)	0.470 (85)
	0.7	94	141	0.678	0.590	0.766 (72)	0.447 (63)
	0.8	57	57	0.718	0.722	0.842 (48)	0.509 (29)
	0.9	41	44	0.709	0.693	0.854 (35)	0.432 (19)
BERT	0.0	589	689	0.561	0.518	0.593 (349)	0.406 (280)
	0.6	144	181	0.611	0.576	0.694 (100)	0.431 (78)
	0.7	94	141	0.639	0.570	0.766 (72)	0.440 (62)
	0.8	57	57	0.676	0.626	0.719 (41)	0.421 (24)
	0.9	41	44	0.647	0.600	0.732 (30)	0.341 (15)

†Embeddings of test cases are always done with code2vec.

*T: Threshold of test case similarity. Given the failing test case of an APR-generated patch, we consider only historical test cases with the similarity which are higher than the threshold. Thus, depending on the threshold, some generated patches cannot be assessed as we are not able to associate them with any past test case.

"(#)" in last two columns represents the number correct/incorrect patches identified by BATS.

classical challenge of search engines [200]. Therefore, we propose to filter in only test cases that present a sufficient level of similarity with the targeted test cases. Experimental evaluations will further offer insights on the use of such a threshold.

We chose the cosine similarity for the baseline and BATS implementation. Table 5.3 reports the classification performance of the baseline (AUC less than 0.6). For reference, the performance of BATS is provided later (cf. Table 5.4). We note that the baseline performance is similar with BATS when the test similarity threshold is not set. However, when we only consider test cases with higher similarity with the failing test cases, the performance of BATS increases (up to 0.85 for +Recall and 0.71 AUC). CC2Vec, as an embedding model for patches, helps achieve high AUC, F1, +Recall (i.e., the recall in identifying correct patches) and -Recall (i.e., the recall in filtering out incorrect patches).

Table 5.5 provides performance results in terms of MAP and MRR. The high metric values further confirm that most correct patches are indeed ranked higher in the recommended patch list that is sorted based on their similarities with historically-relevant patches (given test case similarity). The MAP and MRR of the baseline are both 0.63, underperforming against BATS (up to 0.80 and 0.8 for MAP and MRR respectively).

Table 5.5: BATS's performance on ranking correct patches.

Threshold of test case similarity	0.00	0.60	0.70	0.80	0.90
MAP	0.62	0.63	0.71	0.80	0.70
MRR	0.63	0.65	0.74	0.81	0.75

Figure 5.13 illustrates the overall performance evolution of BATS when the threshold of the test case similarity is varied. We note, while the -Recall does not change drastically, there is a positive effect on +Recall (and other metrics) when the similarity threshold is increased. These results confirm that the underlying

hypothesis of BATS is just: when BATS identifies highly similar test cases, its prediction of correctness for APR-generated patches is more accurate.

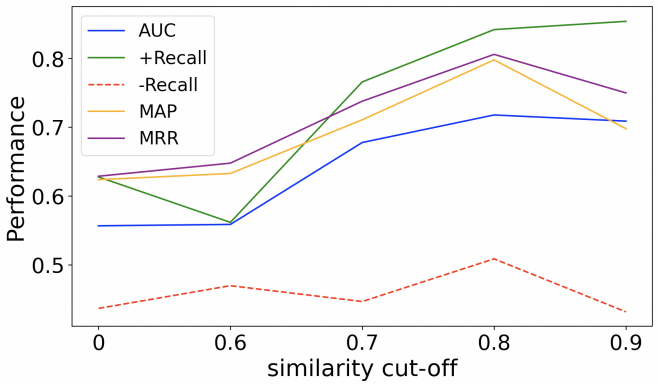


Figure 5.13: Performance evolution of BATS with varying threshold of the test-case similarity.

Representation learning vs. raw strings Beyond the performance metrics on AUC, +/-Recall and F1 of BATS, we propose to investigate the choice of representation learning for embedding patches. To assess the value of leveraging deep learning models for embedding patches, we consider computing similarity of patches as raw strings. To that end, we simply use the Levenshtein distance between the patches (considered as strings, and not after computing embeddings): by setting the threshold of test case similarity as 0.8, we obtain the lowest AUC and F1 values, respectively at 0.46 and 0.53. -Recall even drops at 0.12. These results validate our decision to leverage deep representation learning models for producing patch embeddings.

[RQ-2] *BATS performs well in identifying patch correctness with an AUC ∼ 0.7 and an F1∼0.7 while the Recall of identifying correct patches reaches ∼0.8. Further experimental investigations, with constraints on test case similarities, support our hypothesis: similar test cases are addressed by similar patches.*

5.4.3 RQ-3: Competitive/Complementary to the State-of-the-art

[Objective]: In previous research questions, we validate our hypothesis and develop the pipeline BATS to identify the correctness of patches generated by APR tools. The experimental results show BATS achieves promising performance. To further evaluate BATS in practice, we compare it against the state of the art static and dynamic approaches with the same dataset used for RQ-2. Recall that, in practice, the performance of BATS, is impacted by the availability to find (really) similar test cases. As illustrated in the results of Table 5.4, when the threshold of test case similarity is set high, the number of patches for which similar test cases are found in our dataset decrease significantly. While this does not contradict the hypothesis underlying BATS, it may limit its value when larger datasets are not available. Nevertheless, we postulate that BATS can be complementary to the previous two state of the art approaches (i.e., Tian et al. [3] and PATCH-SIM [2]).

I. Comparing Against the State-of-the-art

We conduct the comparison of BATS against the state-of-the-art patch correctness predicting tools, i.e., Tian *et al.*'s deep learning approach [3], PATCH-SIM [2] and ODS [27].

BATS vs. Deep learning approach

Since BATS leverages pre-trained DL-based (deep learning representation) models, we proceed to compare it against the recent related work by Tian *et al.* [3] where DL-based embeddings of patches are leveraged for static checks. To ensure that the results of BATS and of the DL-based approach are compared on the same dataset, we focus on the 114 patches (threshold>0.8) as the testing set of the DL-based approach. Because the approach of Tian *et al.* require a training dataset for supervisely producing a classifier, we use the remaining 1164 (1278-114) patches. As for classification algorithms, we leverage both Logistic Regression (LR) and Random Forrest (RF), following the experiments of the authors. As shown in in Table 5.6, BATS can recall (+Recall) more correct patches while preserve the same or higher -Recall. It should be noted also that, unlike the DL-approach by Tian *et al.*, BATS doesn't require large dataset for training a classifier.

Table 5.6: Comparison with a state of the art supervised classifier [3].

Classifier	AUC	F1	+Recall	-Recall
Tian et al. (LR)	0.72	0.68	0.77	0.51
Tian et al. (RF)	0.70	0.49	0.75	0.46
BATS	0.72	0.72	0.84	0.51

BATS vs. PATCH-SIM

The state of the art in dynamic assessment of patch correctness is PATCH-SIM [2]. It targets excluding incorrect patches via comparing execution behaviour of tests for the patched and original programs. We apply PATCH-SIM to the 114 test data to generate prediction. However, PATCH-SIM fails to produce prediction results for some of the bugs/patches[7]. Furthermore, we observed that PATCH-SIM takes several hours to produce a prediction outcome for some patches. In our experiment, we set a one hour timeout for the prediction per patch. Eventually, 48 out of 114 could be evaluated by PATCH-SIM. Our experiment was conducted on Linux server equipped with 8 cores 2.10GHz CPU and 125G memory. Results in Table 5.7 show that PATCH-SIM achieves a 0.80 of +Recall and a 0.42 of -Recall. Among patches evaluated, the average prediction time for each patch is 1044s (i.e., more than 17 minutes), while BATS only spends ∼0.3 second on validating each patch. Overall, the dynamic approach, PATCH-SIM, is constrained in terms of resource requirements, as it needs to generate new test inputs and exploit the behavior similarity of test case executions for validating each patch. The resource cost of BATS is mainly decided by two aspects: ❶ the model training process, although BATS leverages the pre-trained models, and ❷ the search space of historical test cases. The time cost of finding similar test cases will be sharply increased only when the search space is expanded.

[7]We reported the problem to the PATCH-SIM authors and we are still waiting for their response.

Table 5.7: Comparison with a state of the art dynamic-based patch assessment [2]

Classifier	AUC	F1	+Recall	-Recall
PATCH-SIM	0.61	0.52	0.80	0.42
BATS	0.72	0.72	0.84	0.51

BATS vs. ODS

We also compare our performance against a recent machine learning-based approach leveraging manually-engineered features. We compare BATS against a recent work by Ye *et al.* [27] where the authors propose a supervised learning approach ODS that explores manually engineering patch features for overfitting detection. To predict patch correctness, ODS first constructs the engineering features from the AST representation of patches to express potential behavior information. Such features are used by ODS to train a ML-based model to proceed the classification of patch correctness. While we could not fully reproduce their work on our dataset due to the unavailability of their training dataset, we are able to compare our results with the ones presented in their paper since we have test sets of similar size and from the same sources. Overall, BATS and ODS exhibit similar performance metrics. When they tune their learners to have a high +Recall (e.g., 1.00), their -Recall drops (e.g., 0.46, respectively). Our BATS unsupervised learning approach further aims to cope with two challenges with approaches such as ODS: (1) they require large sets of labelled patches to perform supervised learning; (2) it can be difficult to manually explain a prediction classification (e.g., because features that contribute to the classification decision are difficult to track back to the failing test case specification and may not generalize to new data).

[RQ-3] ❶ *When the availability of similar test cases is satisfied, BATS can achieve competitive performance on predicting the correctness of APR-generated patches against the state-of-the-art dynamic and static approaches.*

II. Enhancing the State-of-the-art with BATS

We present experimental results to demonstrate that we can achieve enhanced performance in correct patch identification by using existing (static or dynamic) state of the art approaches in conjunction with BATS.

Supplementing a supervised classifier with BATS

[Objective]: Given that BATS is fairly accurate when highly similar test cases (with associated historical patches) are available, we propose to build a pipeline where BATS is applied on the subset of bug cases where such test cases exist. For the rest of bugs, the correctness of generated patches is predicted by using a relevant literature classification-based approach proposed by Tian *et al.* [3]. We then compare the performance of this pipeline against the performance yielded when the classifier is used alone on the whole dataset.

[Experimental Design]: We set a threshold of the test case similarity at 0.6 (cf. Table 5.4) to identify which failing test cases are relevant for assessing the added-value of BATS. The dataset is then split into 325 patches for test and 935 for training a patch classifier described in recent literature: we reproduce the work of Tian *et al.* [3] using their provided artefacts which provide a supervised learning model to classify patches based only on embeddings (computed with Bert). As for

learner, we leverage alternatively Logistic Regression (LR) and Random Forrest (RF) following the experiments of the authors.

We first compute the performance of the supervised learning classifier alone on the test dataset. Then, we evaluate the combined pipeline (Tian *et al.* [3] + BATS) with the following procedure: BATS is first applied to predict correctness for all patches that are associated to test cases for which historical test cases with a high similarity (≥ 0.9) can be found. 26.1% (85/325) of patches in the test set are then evaluated by BATS. The rest of patches are passed to the trained supervised classifier which does not require test case information.

[Experimental Results]: The results presented in Table 5.8 show that the combined pipeline can indeed supplement the baseline classifier. +Recall can be improved by up to 5 percentage points while -Recall can be improved by up to 4 percentage points.

Table 5.8: Supplementing a supervised classifier with BATS.

Classifier	AUC	F1	+Recall	-Recall
Tian et al. (LR)	0.75	0.60	0.53	0.80
Tian et al. (LR) + BATS	0.75	0.62	0.53	0.85
Tian et al. (RF)	0.75	0.59	0.56	0.82
Tian et al. (RF) + BATS	0.75	0.67	0.61	0.82

[RQ-3] ❷ *BATS can supplement a state of the art patch classification system, which : on bug cases where the failing test case is highly similar to historical test cases, BATS can provide more accurate classification.*

II. Complementing test execution based detection

[Objective]: Our objective is to assess whether BATS (which statically reasons about test case similarity) can boost PATCH-SIM [2] (which considers dynamic execution behaviour). We consider that BATS value can be confirmed if it can help exclude incorrect patches that PATCH-SIM could not. Therefore, we build on a similar pipeline than in Section 5.4.3, where BATS is applied instead of PATCH-SIM when enough similar test cases can be found.

[Experimental Design]: We apply PATCH-SIM to the above 325 test patches in last Section 5.4.3. As for BATS, we use the Defects4J dataset as search space. Note that, as in all experiments, we ensure that the search space does not include test case or patches linked to the assessed generated patch. To achieve reliable performance with BATS, we must consider only cases where highly similar test cases exist in our dataset. Therefore it is possible that our performance measurement could only be computed on a portion of the test set. Finally, PATCH-SIM can successfully produce results for 153 patches, of which 48 patches are applied by BATS when setting the similarity threshold at 0.8. is sufficient to find similar test cases.

[Experimental Results]: Table 5.9 presents the performance results of PATCH-SIM (alone) and the combination (PATCH-SIM with BATS). We note that, by applying BATS to the subset of patches where similar test cases are available, we are able to improve the overall performance in patch correctness prediction. Theses results demonstrate that BATS can be complementary to a dynamic approach.

Table 5.9: Complementing PATCH-SIM with BATS.

Classifier	AUC	F1	+Recall	-Recall
PATCH-SIM	0.62	0.55	0.82	0.43
PATCH-SIM + BATS	0.65	0.59	0.84	0.51

[RQ-3] ❸ *BATS static approach is complementary to the PATCH-SIM [2] dynamic approach. By being able to identify incorrect and correct patches when test cases are sufficient, BATS implementation confirms our initial hypothesis that statically reasoning about similar test cases offers a novel and promising perspective to the assessment of APR-generate patch correctness.*

5.5 Ablation Study

5.5.1 Bug types of failing test cases clusters

In this study, we manually check whether bugs, grouped with respect to the test cases, in each cluster are actually similar or not in terms of bug types. We first note that the failing test cases grouped in the same cluster perform similar checks (e.g., date-related checks). Building upon the dissection study of Defects4J bugs by Sobreira*et al.* [201], we further investigate the categories of bugs in each cluster. We find that test cases in a given cluster are indeed related to bugs in the same category. For instance, test cases triggering Chart-2 and Chart-4 bugs are in the same cluster and the dissection study data indicates that these two bugs are in the same category of `NullPointerException`. However, it's important to note that these categories are organized solely based on types of exceptions, rather than fine-grained behavior. As a result, some of our clusters simply fall into the same overarching category. Finally, note that the dissection study was performed in a previous (smaller) version of Defects4J, which does not allow us to provide comprehensive results for our dataset. Nevertheless, the observations that we have made support the framing of our hypothesis that test cases similarity can reflect very well the category of bugs, and hence of the required fixes. In future work, we will consider exploring other possible representations of bugs beyond test cases, such as bug reports.

5.5.2 Asymmetry of the hypothesis

The experimental results validate the feasibility of our proposed hypothesis: similar test cases can lead to similar patches for fixing associated bugs. We investigate the symmetry of the hypothesis: *Could similar patches be referred to similar test cases*? To this end, we first independently cluster patches with their similarities, and then assess the similarities of the associated test cases in each patch cluster. With respect to independent clustering of patches, the clustered patches achieve a Cluster Similarity Coefficient (CSC) of 0.26 and all the groups are qualified. However, the CSC for associated test cases groups is 0.03 that is very close to zero and much lower than the CSC value of clustered patches. The results indicate that the symmetry of the hypothesis is invalidated, i.e., similar patches cannot fully be mapped to addressing similar failing test cases. It is reasonable, since different bugs can be fixed with similar code change ways which lead to similar patches [202], but the different bugs are triggered by different test cases.

5.6 Threats to Validity

Threats to External Validity. We relied on Bisecting-K-means for the clustering experiments to validate our hypothesis. Other algorithms may reveal different results. We have mitigated a potential bias by using multiple evaluation metrics to exhaustively assess the clusters. We also relied only on Defects4j to ensure that we can collect enough plausible patches from literature APR experiments.

In future work, the community could further investigate other datasets as well as test cases augmentation (through test generation [203, 204, 205] or code search [206]) to enlarge the datasets of patches and test cases.

Threats to Internal Validity A major threat to internal validity is that we manually process patches to build the dataset. We may have introduced some mismatching errors when associating test cases. To mitigate this threat, we publicly release all artefacts for review by the community.

Towards reasoning about code similarity, we rely on code2vec, which parses test cases to deep learning features. Unfortunately, while most projects format their test case specifications as for typical code (such as in Figure 5.14), the Closure project presents an ad-hoc format where the essential parts of the specification are formatted as a string (see Figure 5.15): in such cases, code2vec embeddings abstract away the string as a mere argument to a function, rendering the embeddings semantically irrelevant. Therefore, because we leverage off-the-shelf tools to validate our hypothesis, we simply discard Closure bugs, for which future work can investigate specific learners to parse test specification defined in the form of string. Eventually, our experimental evaluation considers 1,278 plausible patches, 598 of which are correct while 689 are overfitting (i.e., incorrect). Overall, the dataset is, to the best of knowledge, the largest set of plausible patches explored in the literature on patch correctness assessment.

```
public void testDrawWithNullInfo() {
   boolean success = false;
      try {
         BufferedImage image = new BufferedImage(200, 100,
         BufferedImage.TYPE_INT_RGB);
         Graphics2D g2 = image.createGraphics();
         this.chart.draw(g2, new Rectangle2D.Double(0, 0, 200, 100), null, null)
    ; g2.dispose();
      }
   catch (Exception e) {
         success = false;
   }
   assertTrue(success);
}
```

Figure 5.14: A typical failing test case specification (Chart-26).

```
public void testComplexInlineNoResultNoParamCall3() {
       test("function f(){a();b();var z=1+1}function _foo(){f()}",
              "function _foo(){{a();b();var z$$inline_0=1+1}}");
```

Figure 5.15: String-based format for test specification (Closure-49).

Threats to Construct Validity. The used embedding models are pre-trained and some parameter weights may not be adapted to our work. In future work, we could retrain and fine-tune the parameters after collecting large datasets.

5.7 Related Work

Representation Learning. Embedding is a key challenge for reasoning about similarity. Initial works on software engineering artefacts have explored models [155, 159] trained on natural language text (such as Wikipedia). BERT is a widely used bidirectional encoder representations model in textual tasks where it outperforms the state of the art by learning to obtain the conditional parameters. Recent works have however proposed specialized architectures to be trained on code. For example, Alon *et al.* [167] proposed code2vec to capture the semantic properties of code snippets by learning distributed representation. Zhangyin *et al.* presented a pre-trained CodeBert [154] that leveraged programming language and natural language data on code tasks. To learn the representation of code changes, Hoang *et al.* [163] proposed an attention-based neural network model CC2Vec with the hierarchical structure to predict the difference between the removed and added code of the patch.

Code Similarity and Code Clone Detection. Finding code that implements similar functionality is what BATS does to find similar test cases and similar patches. DeepSim [157] leverages deep learning on semantic features matrix representing control- and data-flow information to predict whether a given pair of functions implements similar functionality. Fang et al. [158] introduced a new granularity level of the call-callee relationship to capture similar functionality while leveraging word embeddings and graph embeddings to train a deep neural network for the same task.

To detect code clones across different programming languages, CLCDSA [207] extracts 9 syntactic source code features from the ASTs of different pieces of code and uses either Cosine similarity or trained neural network to detect clones. BATS also uses Euclidean distance and Cosine similarity to find similar test cases and similar patches. Alternatively, SLACC [208] uses dynamic analysis and input-output pairs to detect clones across different programming languages. Finally, binary code similarity has been studied extensively [209] with recent approaches also leveraging graph and instruction embeddings [210, 211, 212] to find similar binaries. These approaches for finding similar code across different programming languages and across binaries could benefit BATS in the future by enabling cross-project and cross-language search for similar test cases and similar patches.

5.8 Conclusion

In this work, we propose and investigate a simple yet effective hypothesis for static patch correctness assessment: given a failing test case, any associated generated patch is likely correct if it is similar to patches that were used to address similar failing test cases. We have validated our hypothesis using the developer correct patches and the associated failing test cases in the Defects4J benchmark. To evaluate the potential of this hypothesis in predicting patch correctness, we propose a patch identification system, BATS, to check patch behaviour against test specification based on unsupervised learning. BATS achieves its highest performance when the similarity of test cases is high, which further validates our hypothesis. Comparing against state of the art, BATS outperforms them in identifying correct patches and filtering out incorrect patches. Despite potential issues in the availability of large datasets of projects to search for historical examples (test cases and their associated patches), we demonstrate that BATS can be complementary to both state of the art static and dynamic approaches to predict patch correctness. In summary, BATS highlights a promising avenue for research in patch assessment. It paves the way for future studies focused on delving into deep bug semantics—specifically, the semantics of failing test functions in our context—to establish correlations with correct patches.

6 Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness

In this chatper, we propose a novel perspective to the problem of patch correctness assessment: a correct patch implements changes that "answer" to a problem posed by buggy behavior. Concretely, we turn the patch correctness assessment into a Question Answering problem. To tackle this problem, our intuition is that natural language processing can provide the necessary representations and models for assessing the semantic correlation between a bug (question) and a patch (answer). Experiments on a large dataset of 9 135 patches generated for three bug datasets (Defects4j, Bugs.jar and Bears) show that QUATRAIN *achieves an AUC of 0.886 on predicting patch correctness, and recalling 93% correct patches while filtering out 62% incorrect patches.*

This chapter is based on the work published in the following research paper:

- **Haoye Tian**, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and TegawendÉ F. BissyandÉ. Is this Change the Answer to that Problem? Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1-13. 2022.

Contents

6.1 Overview

Generate-and-validate techniques have achieved success in automatic program repair (APR) by yielding valid patches for a large number of defects in several benchmarks [213, 214, 175, 215, 15, 30]. While such techniques are commonplace, their adoption by the industry faces a critical concern with respect to their practicality: state-of-the-art approaches tend to generate patches that overfit the weak oracles (e.g., test suites) [77, 1, 85, 126]. Indeed, in practice, patches validated by test cases are only plausible. Most of them will be manually found by practitioners to be incorrect [175, 28, 29].

Research on automatic assessment of patch correctness has been prolific in recent years [3, 216, 33, 136]. We identify mainly two categories leveraging either static or dynamic information. In the first category, only static information is leveraged to decide on patch correctness. For example, Ye *et al.* [27] have manually crafted static features of code changes that can be used for training a machine learning (ML) based classifier of patch correctness. Similar approaches based on deep representation learning have been proposed [3]. More recently, Tian *et al.* [4] proposed a system where correctness is decided by checking the static similarity of failing test cases vs the similarity of code changes. In the second category, traces of dynamic execution of test suites are leveraged for correctness evaluation. To predict patch correctness, Xiong *et al.* [2] check the behavioral change of failing test case executions. Shariffdeen *et al.* [56] relied on concolic execution to traverse test inputs and patch spaces to reduce the number of patch candidates.

Despite the promising results achieved by the aforementioned approaches to patch correctness assessment, we identify one fundamental issue and one opportunity that open roads to the new research direction studied in this work. As a fundamental issue, we note that state-of-the-art approaches generally assess patch correctness by reasoning mostly about the code changes, and sometimes also about the test case. However, **the bug itself, which is targeted by the generated patch, is seldom explicitly investigated**. Yet, patches are written to address a specific buggy behavior. As an opportunity, we note that bug reports, while informal, may offer an explicit description of the bug, which can be leveraged to assess patch correctness.

To the best of our knowledge, no prior work has investigated the problem of patch correctness as a question-answering problem. We follow the intuition that when a code base maintainer is presented with a patch, the suggested changes are evaluated with respect to the reported bug. That bug is therefore a question. Bug reports, with their natural language description (cf. Example of Figure 6.1(a)), typically pose the problem. The code changes implementing a patch offer an answer to the problem. The commit message describing these changes (cf. Example of Figure 6.1(b)) typically presents the solutions to the maintainer. The maintainer can then immediately perceive whether the solution (patch) would be relevant to the problem (bug). This scenario of patch validation by human maintainers may appear naive since there are other aspects that developers consider, including whether the bug is real, whether the changes are riskier, etc. Nevertheless, this constitutes a first screening process that we aim to automate by leveraging recent advances in natural language processing (NLP) and machine learning (ML).

This paper. We explore the feasibility of leveraging a deep NLP model to assess

```
Missing type-checks for var_args notation
```

(a) **Title of the bug report.**

```
check var_args properly
```

(b) **The commit message of the developer's patch.**

Figure 6.1: The bug report of Closure-96 from Defects4J and the corresponding commit message of the developer's patch.

the semantic correlation between a bug description and a patch description, towards predicting patch correctness for automated program repair. Our main contributions are as follows:

❶ We perform a preliminary validation study to demonstrate that bug and patch descriptions are correlated within a dataset of developer submitted patches. This hypothesis validation constitutes a first finding that opens a novel direction for patch correctness studies using bug artifacts.

❷ We formulate the patch correctness assessment problem as a question answering problem and propose QUATRAIN (Question Answering for Patch Correctness Evaluation), a supervised learning approach that exploits a deep NLP model to classify the relatedness of a bug report with a patch description.

❸ We extensively evaluate the effectiveness of QUATRAIN to identify correct patches as well as filter out incorrect patches among a dataset of 9,135 plausible patches (written by developers or generated by APR tools). Our evaluation further compares QUATRAIN to state-of-the-art dynamic [2] and static [3] approaches, and demonstrates that QUATRAIN achieves comparable or better performance in terms of AUC, F1, +Recall and -Recall.

❹ We conduct an analysis of the impact of inputs quality on the prediction performance. In particular, we show that the software engineering committee could benefit from extended research into the direction of patch summarization (a.k.a. commit message generation).

Availability. Our artifact, code, and dataset are publicly available at: `https://github.com/Trustworthy-Software/Quatrain`.

6.2 Related Work and Hypothesis

In this section, we describe the related work to highlight the relevance of our work and the novelty of our approach. Then we validate the hypothesis that QUATRAIN builds on.

6.2.1 Related work

Leveraging NLP in program repair. Given that the target of program repair is to transform a buggy program into its correct version, a number of recent studies have considered it as a translation task. Building on the software naturalness hypothesis [164], researchers proposed to apply existing neural machine translation (NMT) techniques generally leveraged in natural language processing. Chen *et al.* [65] proposed a recurrent neural network (RNN) based approach that fixes one-line bugs by translating the buggy line into the correct line. Tufano *et al.* [217] designed another RNN based model that works at the method level: the model takes a buggy method as input and generates the entire fixed method as output. Lutellier *et al.* [194] proposed to separately encode the buggy line and its surrounding contexts (*i.e.,* statements that appear before or after the buggy line). CURE [218], a more advanced approach, leverages pre-training techniques to help the model gain knowledge about the rigorous syntax of programming languages and how developers write code. Another recent study [219] investigated the feasibility of applying a large-scale pre-trained model, CodeBERT, to generate patches.

Overall, while these previous works apply NLP techniques to the patch generation process, our work investigates NLP models for patch correctness assessment.

Leveraging bug reports in software engineering tasks: Bug reports are considered as invaluable resources for debugging activities since they typically contain detailed descriptions about the program failures as well as the clues of the fault (usually in the form of stack traces) [220]. A number of studies have exploited bug reports to facilitate diverse software engineering tasks. Liu *et al.* [221] and Koyuncu *et al.* [107] investigated the feasibility of building program repair systems based on bug reports, instead of the traditional test cases. Indeed, the primary motivation of their works is that the required test case in APR for triggering the bug may not be readily available in practice when the bug is reported. Fazzini *et al.* [222] and Zhao *et al.* [223] explored how to automatically reproduce program failures from bug reports without human intervention. By leveraging code change patterns mined from bug reports, Khanfir *et al.* [224] proposed an approach that injects realistic faults to improve mutation testing. Besides, bug reports have been utilized for constructing high-quality defect benchmarks for software testing [172, 225]. In our study, we leverage bug reports to model the semantics of the bug and thus better assess the patch correctness.

6.2.2 Hypothesis Validation

Our hypothesis is that there is a semantic correlation between a bug description and the associated (correct) patch description. To validate the existence of such a correlation, we conduct a preliminary experiment on a collected dataset. The experiment investigates the semantic similarity between the descriptions. To that end, we consider a ground truth dataset of Defects4J bugs for which a bug report is available and the commit messages describing developer-written patches are provided. These are denoted “original pairs”. Then, we assign a randomly selected commit message

to each bug report in order to build 'random pairs'. Finally, to capture the semantics of the natural language sentences forming the descriptions (of bugs and patches), we utilize a pre-trained deep learning model BERT [155] (introduced in Section 6.3.4), which embeds the descriptions into vectors. We standardize[1] these vector values to eliminate the influence of dimension on the similarity computation. Finally, we calculate Euclidean distance for all pairs. Figure 6.2 presents the distribution for original pairs and random pairs. The results show that the original (*i.e.,* ground truth) bug report and associated commit message pairs are more similar than random pairs. The Mann–Whitney–Wilcoxon test [226] (p-value: 1.2e-32) further validates the significance of the distribution difference. Note that we use semantic similarity as a metric to determine correlation. The validation of the existence of such correlation motivates the QUATRAIN approach, where NLP modelling is leveraged to develop a classification approach of patch correctness by building on predicting the relevance of a patch (based on its description) for a bug (given its description).

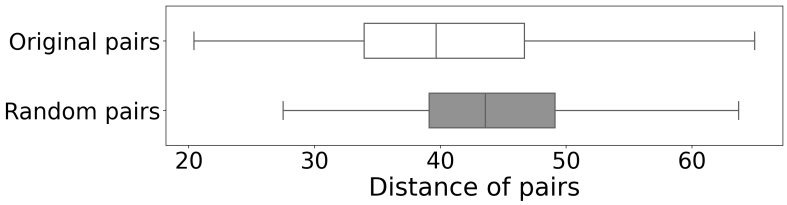


Figure 6.2: Distributions of Euclidean distances between bug and patch descriptions.

[1]In a standardized dataset, each feature has a range of values with a mean of 0 and standard deviation of 1.

6.3 Approach

In this section, we first describe the overview of our proposed approach. Then, we fill in the details of the approach with specific steps separated in several subsections.

[**Overview**]**:** The intuition we build on is that the natural language description of a bug-fixing patch is semantically related to the bug report describing that specific bug: the bug report *describes the problem (bug)* and the patch description *describes the solution to the problem.* This semantic relation between a bug report and its associated patch is similar to the QA relation between questions and their answers in NLP. We present the definition as follow:

Definition 6.3.1 (Patch Correctness Prediction as a QA Problem) *Given a bug report in natural language br_{nl}, a patch $patch_c$ for the reported bug, and a natural language description $patch_{nl}$ of the patch, predict whether the QA-like pair $(br_{nl}, patch_{nl})$ is matching or not. I.e, predict whether the patch $patch_{nl}$ is relevant to (*answers*) the bug report br_{nl} (*the question*) or not.*

To solve this problem, we propose an approach, QUATRAIN, which takes as input a program whose buggy behavior is described in a bug report and the associated patch generated by an APR tool, and outputs a prediction of correctness. Figure 6.3 provides an overview of the approach, which includes two phases: training (offline) and prediction (online).

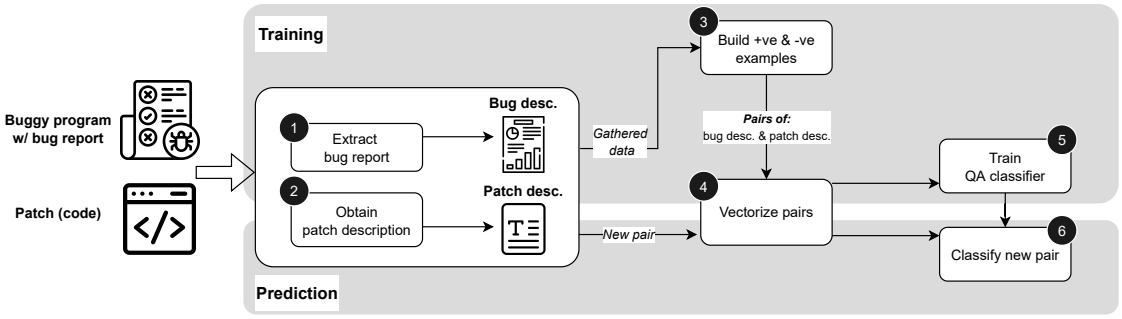


Figure 6.3: Overview of the approach.

In the training phase, given a batch of a buggy program, QUATRAIN first extracts the bug NL description (bug report) from the program repository in an automatic way; then, for each candidate patch associated to the bug, it generates the patch NL description by leveraging a code change summarization tool (e.g., a commit message generator - cf. Sections 6.3.1 and 6.3.2). Subsequently, QUATRAIN requires a large number of positive (i.e. correct) and negative (i.e. incorrect) examples to train a classifier that predicts the correlation between a bug report and a patch description. Our third step (Section 6.3.3) thus focuses on building a dataset of positive and negative examples of pairs of bug reports and associated (in)correct patches. In the fourth step (Section 6.3.4), QUATRAIN converts the patch descriptions and the bug reports into vectors in high-dimensional vector space to enable model learning. Finally, in the fifth step (Section 6.3.5), QUATRAIN trains a neural QA classifier on the pairs of bug reports and patch descriptions.

In the prediction phase, QUATRAIN pre-processes a new buggy program and its associated candidate patch by applying the first, second, and fourth steps in Figure 6.3. It then uses the trained QA classifier to predict whether the candidate patch indeed answers the problem in the bug report. The answer is equivalent to a

statement on the correctness of the plausible patch (yes:correct; no:incorrect).

6.3.1 Extraction of Bug Reports

The first step of our approach is to obtain descriptions of the bugs. A natural choice for finding such descriptions is to leverage bug reports. They exist in large numbers across projects and provide a NL description of program buggy behavior which, at least, describes the symptom of the bug. Bug reports are submitted via different platforms, e.g., issue trackers such as Jira and issues in GitHub. For our purpose, we use a script to automatically mine bug reports for the bug datasets that we use.

An official bug report typically includes three parts: title, description, and comments. In the benchmark that we build, some bug reports include comments where the correct solution or even the entire patch is posted. Note, however, that in our experimental assessment, we must assume that the bug has not yet been fixed. Thus, to remain practical and reduce bias, we discard all comments and leverage only the title and the description body of bug reports.

6.3.2 Generation of Patch Description

The second step of our approach is to summarize an APR-generated patch in natural language so as to obtain a semantic explanation of the changes applied in the patch. The idea here is to get a representation of the patch that is as close as possible to how a bug report describes, in natural language, what is the bug. If the patch is written by a developer (e.g., positive example patches in our training set), its associated commit message could be used as a proxy for such NL description of the patch. We use a script to mine the commit messages from developer repositories such as GitHub and collect the descriptions associated to the patches in our datasets.

Note however that commit messages are obviously not available for APR-generated patches. Therefore, we automatically generate patch descriptions with the help of state-of-the-art commit message generation techniques. In particular, we consider CodeTrans [227], an encoder-decoder transformer based model that has been developed to tackle several software engineering tasks. Quatrain uses CodeTrans-TF-Large, the largest such model which achieves the highest BLEU score so far of 44.41 on the commit message generation task.

During training, we obtain patch descriptions either from: (i) Manually written commit messages of bug-fixing patches provided by developers, or (ii) Automatically generated descriptions using CodeTrans for APR-generated patches.

6.3.3 Construction of Training Examples

Recall from Definition 6.3.1 that we are addressing a binary classification problem. To train a binary classifier, one needs to collect positive (i.e. correct) as well as negative (i.e. incorrect) examples. Therefore, the third step of Quatrain is to build a dataset of positive and negative training examples.

At a high level, a positive (or negative) training (or testing) example consists of a bug report and its associated patch.

Definition 6.3.2 (Bug report-patch description pair) *A* bug report-patch description *pair is a tuple* $(br_{nl}, patch_{nl})$ *of a bug report* br_{nl} *and a patch description* $patch_{nl}$ *(in NL) of a patch that is intended to fix the bug reported in* br*.*

I. Positive Examples

We collect two kinds of positive (correct) training examples. The first kind of correct examples are developer-written patches and their associated bug reports. The second kind of correct examples are APR-generated patches and their associated bug reports where the APR patches have been manually labeled in previous studies [4, 27, 145, 14]

II. Negative Examples

We need to create negative examples to train QUATRAIN to identify incorrect patches. To do so, we build two kinds of incorrect examples.

For the first kind of negative examples, we randomly assign developer-written patches to irrelevant bug reports. For example, we create a training sample by assigning the patch for bug-x with the bug report of bug-y. The rationale for creating this kind of negative examples is to mobilize the model to learn the hidden relations between bug reports and their associated patch descriptions. A patch that tackles a totally irrelevant bug would carry much less - if any - relation to the bug under examination.

The second kind of negative examples is created by selecting APR-generated patches that have been labeled as incorrect in previous studies [4, 27, 145, 14] and their associated bug reports. The idea is that those APR-generated patches were intended to address the specific bug under examination, but a manual verification revealed that they were incorrect. Correspondingly, the patch description generated for such incorrect patch does not correctly answer the bug report and thus could be considered as negative example.

6.3.4 Embedding of Bug Reports and Patches

To efficiently learn the relationship between bug reports and patch descriptions, we first need to convert the text into a numerical representations. Though there exist various techniques [228, 229, 230, 231] for transforming texts into numerical vectors, selecting the proper embedding technique is crucial, as it influences how precisely the vectors represent the text. Compared with popular embedding models such as Word2Vec [168], which uses a fixed representation for each word regardless of the context within which the word appears, BERT [155] has more advantages for representing texts: it produces word representations that are dynamically informed by the words around them. Thus, we employ BERT as our initial embedding model for both bug reports and patch tokenized texts. The used model is a pre-trained large model with 24 layers and 1,024 embedding dimension trained on cased English text. After representing texts into a vector space, we can perform numerical computations on them, e.g., compute text similarity or correlation metrics.

6.3.5 Training of the Neural QA-Model

QA-Models are widely applied in Natural Language Processing (NLP) and Software Engineering (SE) communities. Existing literature on QA has addressed various tasks [232, 233, 234], among which, the task of answer selection shares fundamental similarities with our bug report-patch description matching problem, *i.e.,* select a

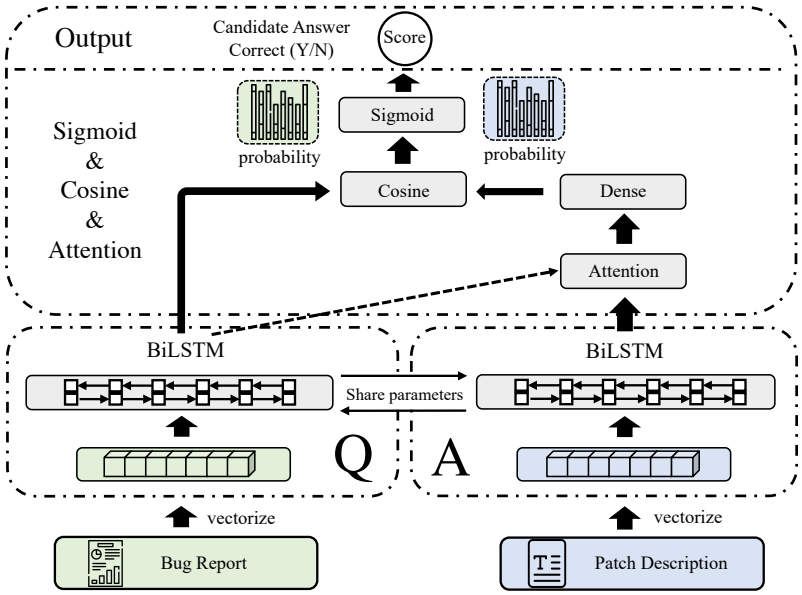


Figure 6.4: Architecture of the neural QA model.

correct answer (patch) for a question (bug report). Tan *et al.* [235] 's approach exhibits powerful performance on this task by extending basic bidirectional long short-term memory (BiLSTM) model with an efficient attention mechanism. Thus, in the fifth step of QUATRAIN, we propose to adapt the extended QA model from Tan *et al.* to learn the correlation between bug reports and patch descriptions. We present the architecture of our adapted QA model in Figure 6.4.

The QA model requires two inputs: bug report and patch description, in their vectorized format (as per Section 6.3.4). Then, the BiLSTM layer takes the inputs to learn the correlation between the bug report and the patch description. Assuming input vectors are $vector_b$ and $vector_c$, we present the BiLSTM in Equation 6.1.

$$\begin{aligned} e_b &= BiLSTM(vector_b) = [xb_1, xb_2, ..., xb_N] \in \mathcal{R}^{N \times dim} \\ e_c &= BiLSTM(vector_c) = [xc_1, xc_2, ..., xc_N] \in \mathcal{R}^{N \times dim} \end{aligned} \tag{6.1}$$

where e_b and e_c represent BiLSTM embeddings of one bug report b and one associated patch description c. N is the length of the input sequence and dim refers to the dimension size of each sequence. xb_i and xc_j are the embeddings of i-th word in b and c.

To better distinguish the correct patch from other patches based on the bug report, we employ an attention mechanism on the patch description to combine the most relevant information according to the bug report, similar to Tan *et al.* [235].

To this end, for each word embedding xc_j in patch description, we compute the matrix product $e_b(xc_j)^T$. We then propagate the resulting vector through a softmax operator to obtain the impact weight α_{xc_j} of each word of bug report to xc_j.

$$\alpha_{xc_j} = Softmax(e_b(xc_j)^T) \in \mathcal{R}^N \tag{6.2}$$

where $\text{Softmax}(x) = \frac{exp(x)}{\Sigma_i exp(x)}$, and *exp(x)* is the element-wise exponentiation of the vector x. Afterwards, we map the impact weight α_{xc_j} back to each bug report word embedding xb_i to obtain attention representation att_{xc_j},

$$att_{xc_j} = \Sigma_n^N \alpha_{xc_j,n} xb_i \in \mathcal{R}^{dim} \tag{6.3}$$

where $\alpha_{xc_j,n}$ means the *n-th* value of α_{xc_j}. Then, we flatten e_b and att_{xc_j} to one-dimensional vectors re_b and re_c representing the bug report and patch description (with attention), respectively.

Finally, we compute cosine similarity between bug report vector re_b and associated patch description vector re_c and use the sigmoid activation function to normalize the output value of cosine layer to the value range of 0 and 1.

$$Score = Sigmoid(cosine(re_b, re_c)) \tag{6.4}$$

where Sigmoid(x) = $\frac{1}{1+exp(-x)}$. The *Score* is the prediction probability of patch correctness.

[Hyper-parameters]: The employed QA model is mainly based on BiLSTM. We set the max sequence length to 64 and the hidden state dimension size to 16 for the BiLSTM layer. During the training period, we iterate the model parameters by using an Adam optimizer with a leaning rate of 0.01. Considering the data size, we execute 10 training epochs to ensure the convergence of the model. The batch size at each epoch is 128.

6.3.6 Classifying a Pair of Bug Report and Patch

For a given buggy program and its APR-generated patch, QUATRAIN classifies the pair as being correlated or not by first extracting the bug description, generating a textual description of the patch, vectorizing the pair of texts, and finally querying the trained QA model. A prediction probability is a value between 0 (incorrect) and 1 (correct). QUATRAIN labels a patch as being correct or not based on a threshold on the prediction output (Section 6.5.1).

6.4 Study Design

We first enumerate the research questions that we investigate to assess the effectiveness of our approach. Then, we describe the dataset used for answering the questions. Finally, we present the evaluation metrics used in our study.

6.4.1 Research Questions

- **RQ-1:** *What is the effectiveness of* QUATRAIN *in patch correctness identification based on correlating bug and patch descriptions?*
  We evaluate QUATRAIN on a large dataset consisting of ground truth correct and incorrect patches.
- **RQ-2:** *To what extent does the quality of the bug report and of the patch description influence the effectiveness of* QUATRAIN*?*
  We perform two separate experiments: in the first, we consider the size of texts (*i.e.,* number of words) as a proxy for quality, and we investigate whether there is a difference in quality measurement across correct and incorrect predictions. In the second experiment, given the original bug report and developer patch description pairs, we replace them alternatively with a random bug report or a tool-generated patch description and observe changes in performance measurements.
- **RQ-3:** *How does* QUATRAIN *perform in comparison with the state-of-the-art techniques for patch correctness identification?*
  We propose to compare our approach against static and dynamic approaches proposed in the literature for APR patch assessment.

6.4.2 Datasets

In this paper, we leverage benchmarks that are widely used in the program repair community and on which several APR tools have been applied to generate a large number of patches: Defects4J [88], Bugs.jar [172] and Bears [173]. Table 6.1 summarizes the patch dataset that we use for our experiments. First, we mainly collect the labeled patches (including developer patches) from the studies of Tian *et al.* [4] and Ye *et al.* [27]. We then supplement the dataset with the patches generated by AVATAR [145] and DLFix [14], which were not considered in these prior works. Considering that different APR tools may generate the same patches for the same bug, we use a simple string-based comparison script to deduplicate our patch dataset. Overall, we obtain a large duplicated patch assessment dataset of 11,352 patches consisting of 2,260 correct and 9,092 incorrect patches. Nevertheless, although we removed some duplicated patches, there are some semantically equivalent patches that could not be detected with our script. For instance, the two conditional statements `if (dataset == null)` and `if ((dataset) == null)` in Java are equivalent, although the extra parentheses make their raw strings mismatch. To reduce the bias of these duplications in our experiments, we design a specific dataset split scheme in Section 6.5.1.

In our experiment: We recall that our approach relies on measuring the correlation between the bug report (BR) and the patch description to predict patch correctness. The collected patches above involve 1,932 unique bugs To obtain the associated bug reports, we mined their code repositories. Unfortunately, 631 bugs do not contain associated bug reports. Eventually, we were able to leverage 1,301 bug reports. Finally, for the 1,301 unique bugs, we obtain 9,135 available patches consisting of 1,591 correct and 7,544 incorrect patches for our experimental evaluation.

Table 6.1: Datasets of labeled patches.

Benchmark	Subjects	Correct	Incorrect	All
Defects4J [88]	Tian *et al.* [4]	1,344	1,017	2,361
	Ye *et al.* [27]	0	5,493	5,493
	AVATAR [145], DLFix [14]	59	38	97
Bugs.jar [172]	Ye *et al.* [27]	930	2,254	3,184
Bears [173]		251	531	782
Total		2,584	9,333	11,917
Total (deduplicated)		2,260	9,092	11,352
Total (experiment)		1,591	7,544	9,135

6.4.3 Metrics

Our objective in patch correctness identification is to recall as many correct patches while filtering out as many incorrect patches as possible. Thus, we follow the definitions of **Recall** proposed by Tian *et al.* for the evaluation of their BATS [4]:

- **+Recall** measures to what extent correct patches are identified, i.e., the percentage of correct patches that are identified from all correct patches.
- **-Recall** measures to what extent incorrect patches are filtered out, i.e., the percentage of incorrect patches that are filtered out from all incorrect patches.

$$+Recall = \frac{TP}{TP + FN} \qquad (6.5) \qquad -Recall = \frac{TN}{TN + FP} \qquad (6.6)$$

where TP represents true positive, FN represents false negative, FP represents false positive, TN represents true negative.

Area Under Curve (AUC) and F1. We construct a deep learning-based NLP classifier to identify the correctness of the patch. Therefore, we use the two most common metrics, AUC (i.e., the overall ability to distinguish between correct and incorrect patches) and F1 score (harmonic mean between precision and recall for identifying correct patches), to evaluate the overall performance of our approach [236].

6.5 Experiments and Results

We conduct several experiments to answer our research questions. In Sections 6.5.1 and 6.5.2, we focus on the evaluation of approach performance and analysis of approach input to performance. In Section 6.5.3, we compare against the state-of-the-art approaches.

6.5.1 RQ-1: Effectiveness of Quatrain

[Experiment Goal]: We answer RQ-1 by investigating to what extent the QUATRAIN approach, which predicts patch correctness by correlating bug and patch descriptions, is effective.

[Experiment Design]: In the literature, ML-based approaches to patch correctness identification are commonly evaluated using 10-fold cross validation (*i.e.,* patch set is divided into 90% for training and 10% for test) [3]. However, as we noted in the analysis of our datasets, there are semantically equivalent patches. Thus the training and testing set may contain duplicate samples, which could lead to biased [2] experimental results due to data leakage (*i.e.,* the model already sees some same test samples in the training phase).

Given the challenge to fully deduplicate the dataset, we propose to limit the bias via a new split scheme, referred to as *10-group cross validation*. A first manual analysis has shown that the duplicated patches are typically generated by different APR tools while targeting the same buggy program. Therefore, we first randomly distribute 1,301 unique bugs (including 9,135 patches) into 10 groups: and every group contains unique bugs and their corresponding patches. Then, 9 groups are used as train data and the remaining one group is used as the test data. Finally, we repeat the selection of train and test groups for ten rounds and average the metrics obtained across the different experimental rounds. Through this 10-group cross validation scheme, each patch is able to be leveraged as train data and test data once, which fits the objective of cross-validation. Additionally though, during each train-test process, the unique bugs along with their sets of semantically equivalent patches are exclusively assigned to either train or test group. We trust that such a scheme will provide a realistic evaluation of the performance of learning-based approaches for patch correctness assessment.

Figure 6.5 shows the distribution of the number of patches assigned to train and test data at each round of 10-group cross validation. The overall ratio of train and test data splits is around 10:1. This ratio is close to typical 10-fold cross validation (9:1) and thus is appropriate to evaluate the performance of train-test based approaches.

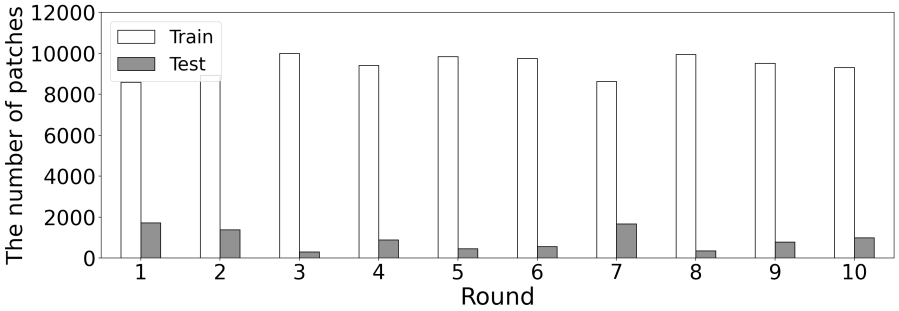


Figure 6.5: Distribution of Patches in Train and Test Data.

[2]We discuss this threat in Section 6.6.

Table 6.2: Confusion matrix of QUATRAIN prediction.

AUC	F1		Thresholds								
			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.886	0.628	#TP	1,591	1,582	1,551	1,475	1,175	583	189	0	0
		#TN	0	2,388	3,010	4,653	6,566	7,261	7,522	7,544	7,544
		#FP	7,544	5,156	4,534	2,891	978	283	22	0	0
		#FN	0	9	40	116	416	1008	1,402	1,591	1,591
		+Recall(%)	100	99.4	97.5	92.7	73.9	36.6	11.9	0	0
		-Recall(%)	0	31.7	39.9	61.7	87.0	96.2	99.7	100	100

[Experiment Results]: Using the presented 10-group cross validation, we provide the overall confusion matrix as well as the average +Recall (recall of correct patches) and -Recall (recall of incorrect patches) of QUATRAIN in Table 6.2.

QUATRAIN achieves high AUC at **0.886**, demonstrating the overall effectiveness of the QA model for patch correctness prediction. We note however that the F1 score (0.628) is relatively low. This metric is known to yield low values when the test data is imbalanced [237]: in our setting, the ratio is around 5:1 between the incorrect and the correct patch sets. We indeed confirm that that better F1 can be obtained by re-balancing the test data: with a ratio of 1:1 (1,591:1,591) at each round, the same trained classifier achieves a F1 score of **0.793**. Later, in our experiments, we mitigate the potential imbalance bias by comparing against state-of-the-art approaches on the same experimental settings (cf. Section 6.5.3). We found that +Recall and -Recall are sensitive to the selection of thresholds. When setting the threshold at a low value (*e.g.,* 0.1), we are able to identify all correct patches (+Recall=100%) but conversely none of the incorrect patches can be filtered out (-Recall=0%). Similarly, at the threshold value of 0.9, we filter out all incorrect patches but cannot recall any correct patch. Nonetheless, we see that QUATRAIN achieves promising results balanced between +Recall and -Recall when an adequate threshold is selected. For instance, QUATRAIN can recall 92.7% correct patches while filtering out 61.7% incorrect patches at a threshold value of 0.4 or +Recall of 73.9% and -Recall of 87.0% respectively at a threshold of 0.5. The results demonstrate our approach is effective on identifying correct and incorrect patches.

✍ **RQ-1** *Experimental validation on our collected ground truth demonstrates the effectiveness of QUATRAIN in identifying correct patches and filtering out incorrect patches: our implementation achieves a +Recall of 92.7% and -Recall of 61.7% when the decision threshold is set at 0.4.*

6.5.2 RQ-2: The Impact of Input Quality on Quatrain

[Experiment Goal]: QUATRAIN relies on specific steps to extract bug and patch descriptions once a patch candidate is generated to be applied for a buggy program. The quality of these descriptions may thus influence the performance of our approach. We investigate such an influence by attempting to answer three sub-questions:

- **RQ-2.1** *To what extent does the length of bug reports and patch descriptions influence the prediction performance?* We hypothesize that good descriptions should have more distinct words, and explore whether correct predictions are made on patch/bug descriptions of larger size.
- **RQ-2.2** *Does the NLP-based QA classifier actually correlate the bug report and the patch description?* We introduce noise in the test data and evaluate whether the classifier is actually looking at the correlation that we seek to check with the QA.

- **RQ-2.3** *Do generated patch descriptions provide the same learning value as developer-written commit messages?* We perform experiments where ground truth patch descriptions in the training set are alternatively switched between developer-written (assumed of high quality) and automatically generated (assumed of lower quality) commit messages.

[Experiment Design (RQ-2.1)]: We first define the length of input sentence as the number of distinct words included in the bug reports. Our assumption is that the presence of more distinct words in a textual description may indicate higher quality. Then, for each evaluation round of the 10-group cross validation scheme, we compute the boxplot distribution of length of bug and patch description for correct and incorrect predictions made by our model respectively. Finally, we calculate Mann Whitney Wilcoxon (MWW) to evaluate whether the difference of length is significant across the distributions. The analysis is made on both the length of bug report and patch description.

[Experiment Results (RQ-2.1)]: Figure 6.6 presents the distributions of patch description lengths for each round of prediction. We observe that, overall in most groups, the length of patch description are bigger in the correct predictions than in the incorrect predictions: the model is effective when the patch description has larger size. The Mann–Whitney–Wilcoxon test (p-value: 4.1e-16) further confirms that the difference of length is statistically significant. In contrast, the difference for the case of bug reports was not found to be statistically significant.

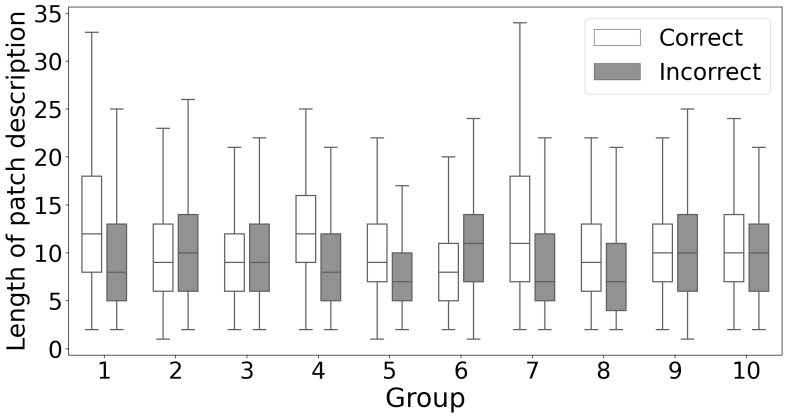


Figure 6.6: Impact of length of patch description to prediction.

✍ **RQ-2.1** *The higher the quality of patch description (i.e., in terms of text length), the more* QUATRAIN *is accurate in predicting patch correctness.*

[Experiment Design (RQ-2.2)]: We recall that our NLP model is designed to correlate the bug report and patch description to predict the patch correctness. To validate that some correlation is indeed learned by the devised model, we investigate the influence of associating wrong bug reports to some patches in the test set. We consider the dataset of 1,301 developer-written patches in this experiment since the developer patch description and associated bug report are known to be indeed related by construction. We first compute the performance achieved by QUATRAIN in the prediction of correct patches. Then, for the patches that QUATRAIN correctly predicts (recall), we re-run the classification test where we replace the original bug reports with other randomly selected bug reports among the test data. We investigate whether this breakdown of the correlation between bug report and patch description

is reflected in the prediction performance of NLP model.

[**Experiment Results (RQ-2.2)]:** Figure 6.7 presents the distribution of prediction probability of QUATRAIN for the 1,073 correct patches when the classifier is applied on the ground truth pairs (*i.e.,* original pairs) and when the classifier is applied on pairs where the patch is associated to a random bug report (*i.e.,* random pairs) . As we see from the boxplot, the lowest value of the distribution of original pairs (white box) is around 0.5. This is normal by construction: we set 0.5 as the threshold probability for deciding correctness, and our data is focused on cases where the prediction was correct. After breaking the correlation of bug report and patch description pairs, we found that QUATRAIN yields some prediction probability values smaller than 0.5 (*i.e.,* they will be wrongly-classified as incorrect) although the patches are correct. The Mann–Whitney–Wilcoxon test (p-value:4.0e-35) confirms that the difference of median probability values is statistically significant between the two distributions. Concretely, 22% (241/1,073) of developer patches, which were previously predicted as correct, are no longer recalled by QUATRAIN after they have been associated to a random bug report. These results suggest that QUATRAIN indeed assesses the correlation between the bug report and the patch description for predicting correctness.

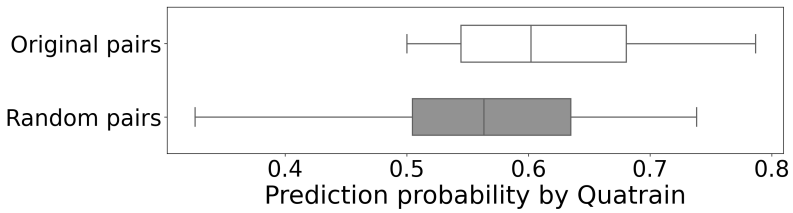


Figure 6.7: The distribution of probability of patch correctness on original and random bug report.

✍ **RQ-2.2** *When developer patches are paired with random bug reports, QUATRAIN is no longer able to predict over 20% of them correctly. The results suggest that the QA learner in QUATRAIN indeed assesses the correlation between the bug report and the patch description for predicting correctness.*

[**Experiment Design (RQ-2.3)]:** Commit messages are generally accepted as high-quality descriptions of changes since they are manually written by Developers. While CodeTrans is a state-of-the-art, its generated-descriptions should be lesser quality. Nevertheless, because developer-written commit messages are unavailable in practice for APR-generated patches, we must resort to automatic patch summarization tools such as CodeTrans. We evaluate the impact of the quality of patch description (developer-written vs. CodeTrans-generated) on the prediction performance. Our experiments focus on the developer patches only as in RQ-2.2. In the dataset, each patch has two kinds of descriptions, *i.e.,* written by developer and generated by Codetrans. We first evaluate our approach based on developer-written descriptions. Then, we replace the developer descriptions with CodeTrans-generated descriptions to assess the performance evolution.

Besides, we speculate that QUATRAIN is more likely to correctly predict a correct patch if the generated description is similar to developer-written descriptions used in the training set, we conduct experiments to validate this hypothesis. Note however that the semantics of developer-written and generated descriptions should be equivalent as they describe the same developer patch. To measure the differences

in the descriptions, we adopt the Levenshtein distance [3] and compute their textual similarity.

[Experiment Results (RQ-2.3)]: The experimental results show that QUATRAIN achieves a +Recall of 82% (1,073/1,301) when the input for test data uses developer-written descriptions of patches as in RQ-2.2. However, that metric (+Recall) drops by 37 percentage points to 45% when the developer-written descriptions are replaced with CodeTrans-generated descriptions. This demonstrates that the quality of patch description considerably impacts the prediction performance of QUATRAIN. Figure 6.8 displays the boxplot distribution of Levenshtein distance between developer description and generated description on correct and incorrect predictions respectively. In most of the groups, the white box (correct predictions) presents the shorter Levenshtein distance value, *i.e.,* higher similarity. This result suggests that if a generated description has a quality that is as high as that of the developer description, QUATRAIN prediction ability will benefit from it. Finally, note that in Section 6.5.1, we evaluated QUATRAIN in a setting where all developer commit messages were replaced with generated descriptions: the AUC metric dropped by 11 percentage points to 0.774, confirming our findings.

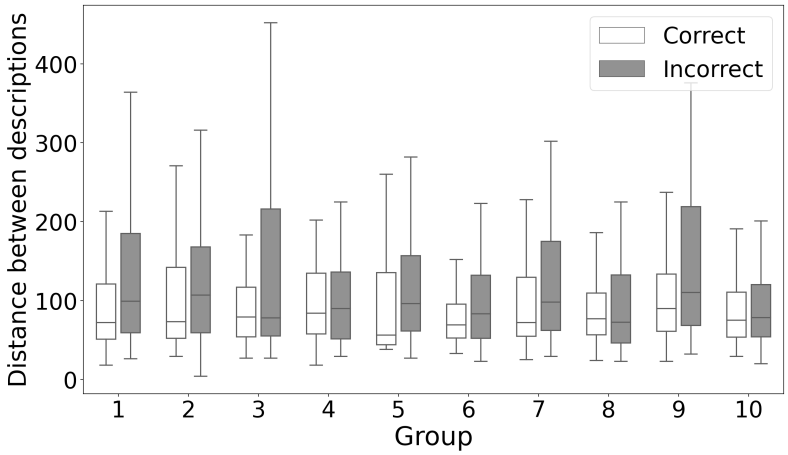


Figure 6.8: Impact of distance between generated patch description to ground truth on prediction performance

✍ **RQ-2.3** *Patch descriptions generated by CodeTrans are often of different quality than ground-truth descriptions. Good patch descriptions help QUATRAIN identify more correct patches.*

6.5.3 RQ-3: Comparison against the State-of-the-art

While previous RQs have shown that QUATRAIN is effective, in this section we compare it against state-of-the-art static and dynamic approaches. Finally, we investigate the complementarity of QUATRAIN to other existing approaches.

I. Comparing against Static Approaches

We compare QUATRAIN against two state-of-the-art approaches: (i) A pure classification approach based on patch embeddings [3] and (ii) BATS which leverages the embedding of test cases to compute similarity among failing test cases and among associated patches [4].

[3]A classic metric for measuring the distance between two strings by calculating the minimal edit operations required.

Quatrain vs. (supervised) DL-based Patch Classifier

In Quatrain, we first leverage pre-trained Bert model to embed the natural language text of bug report and patch description of patch. Then, we build a deep learning classifier to capture the QA relationship between these descriptions to predict patch correctness. Since Tian *et al.*'s approach also use BERT and construct a classifier for patch correctness validation, we compare our approach against theirs. For a fair comparison, we reproduce their evaluation on our dataset. Concretely, when we train or test our model with divided-by-group patches, we consistently use the same patches for the training and testing of Tian *et al.*'s classifiers of Logistic Regression (LR) and Random Forrest (RF), following their experimental setup.

Table 6.3 presents the comparison results: Tian *et al.*'s best classifier (RF) achieves +Recall of 89.4% while filtering out 59.8% incorrect patches. Meanwhile, Quatrain achieves a better +Recall of 92.7%, and filters out slightly more incorrect patches (-Recall of 61.7%). Regarding the overall performance metrics AUC and F1, Quatrain outperforms the approach of Tian *et al.*. We finally investigate the complementarity of our approach. Among 9,135 patches, our approach identifies 7,842 patches, of which 2,735 patches cannot be identified by Tian *et al.*'s approach (RF).

Table 6.3: Quatrain vs a DL-based patch classifier [3].

Classifier	Incorrect:Correct	AUC	F1	+Recall	-Recall
Tian et al. (LR)	7,544:1,591 (5:1)	0.719	0.449	0.833	0.605
Tian et al. (RF)	7,544:1,591 (5:1)	0.746	0.470	0.894	0.598
Quatrain	7,544:1,591 (5:1)	0.886	0.628	0.927	0.617

Quatrain vs. (unsupervised) BATS

BATS [4] is the most recent patch correctness assessment approach proposed by Tian *et al.*. It is devised based on a simple but novel hypothesis that when different defective programs fail to pass similar test cases, it's likely that the programs can be repaired by similar code changes. Given a buggy program, failing test cases and a plausible patch, BATS first searches the most similar failing test cases from other oracle programs. Afterwards, the associated correct patches that fix these similar test cases are extracted to compute their similarity with the generated plausible patch. BATS labels the plausible patch as correct if that similarity is beyond an inferred threshold, otherwise it is predicted as incorrect.

According to the authors' open-source artifacts, BATS is currently able to be evaluated on Defects4J and is not adapted for Bears and Bugs.jar. We thus conduct the comparison on the benchmark of Defects4J. Although Tian *et al.* demonstrated that BATS shows promising results on identifying patch correctness, its scalability is limited due to the lack of enough test cases in the search space to compute similarity. They thus added a cut-off on the similarity computation of test cases to focus on a subset of patches where BATS is applicable. We follow their experimental setup to reproduce BATS evaluation on our dataset with the cut-off of 0.0 (non-specific scenario) and 0.8 (the best performance in their evaluation). We compare our approach on the same available dataset.

As shown in Table 6.4, the configuration of cut-off incurs the reduction of patch set that can be evaluated. Our approach comprehensively outperforms BATS whether they filter dissimilar programs or not (cut-off: 0.0 and 0.8). Note that BATS is not

able to scale its performance, in this scenario, to the entire dataset due to the lack of similar test cases. In addition, in this scenario, 180 out of 345 patches are exclusively identified by QUATRAIN.

Table 6.4: QUATRAIN vs BATS [4].

Classifier	Incorrect:Correct	AUC	F1	+Recall	-Recall
BATS (cut-off: 0.0)	4,930:385 (13:1)	0.549	0.149	0.647	0.452
QUATRAIN	4,930:385 (13:1)	0.824	0.350	0.803	0.662
BATS (cut-off: 0.8)	367:41 (9:1)	0.620	0.235	0.805	0.436
QUATRAIN	367:41 (9:1)	0.832	0.462	0.902	0.453

II. Comparing against a Dynamic Approach

We consider a dynamic approach where execution traces are also leveraged in the prediction of correctness. PATCH-SIM [2] is a state-of- the-art tool for dynamic assessment of patch correctness: it compares test execution information before and after patching a buggy program. The hypothesis they proposed is that correct patches tend to change the behavior of execution of failing test cases and retain the behavior of passing test cases. Due to the failure of prediction for part of the patches [4] and limitation of timeout, we can apply PATCH-SIM to 3,546 patches. The results in Table 6.5 show that our approach filters out more incorrect patches while reaching same +Recall compared to PATCH-SIM. Most of the patches (1,856/3,149) that we identify are not correctly predicted by PATCH-SIM. Note that the low values of F1 score (both for PATCH-SIM and QUATRAIN) are due to the extremely imbalanced ratio of 44:1 in incorrect:correct sets.

Table 6.5: QUATRAIN vs (execution-based) PATCH-SIM [2].

Classifier	Incorrect:Correct	AUC	F1	+Recall	-Recall
PATCH-SIM	3,468:78 (44:1)	0.581	0.053	0.769	0.392
QUATRAIN	3,468:78 (44:1)	0.792	0.127	0.769	0.667

✍ **RQ-3** *Comparing against state-of-the-art static and dynamic approaches, QUATRAIN achieves competitive (or better) performance in predicting patch correctness.*

[4]We reported the problem to the PATCH-SIM authors and we are still waiting for their response.

6.6 Discussion

We enumerate a few insights from our results and discuss the threats to the validity of our study.

6.6.1 Experimental Insights

[*Insufficient deduplication of semantically-equivalent patches may lead to biased prediction performance.*] As we mentioned in the experimental design in Section 6.5.1, the traditional 10-fold cross validation scheme may assign the same semantically-equivalent patches simultaneously into both train and test datasets. In practice, this setup violates the principles in machine/deep learning-based evaluations since it's equivalent to letting the models cheat by learning knowledge from test data during the training process [238, 239, 240]. To showcase this bias in the results, we propose to focus on a straightforward classifier using a random forest on the embeddings of the bug report and the patch: when using 10-fold cross validation scheme on our ground truth dataset, the achieved AUC is as high as 0.978 (with F1 at 0.860); however, when using our deduplication scheme (10-group cross validation based on bug ID), the AUC drops to 0.780 (and F1 at 0.344).
[*Generating high quality code change description can help identify patch correctness*] We found that the quality of code change description influences the prediction performance of QUATRAIN. In RQ-2.1 and RQ-2.3, the experimental results show the model makes more correct predictions when addressing longer or more developer written-similar code change description. Our experiments offer some evidence to encourage the community to design advanced patch summarization approaches. QUATRAIN indeed can become a prime candidate for leveraging such research output to further increase the practicality and adoption of automated program repair.

6.6.2 Case Study

Figure 6.9 presents an example correct patch generated by DLFix, an APR tool, for Defects4J bug Lang-7. QUATRAIN successfully predicts its correctness while BATS fails to do so. The associated bug [5] is reported as follows:

```
Title: NumberUtils#createNumber - bad behaviour for leading "--".
Description: NumberUtils#createNumber checks for a leading "--"
in the string, and returns null if found. This is documented ...
```

```
--- ./src/main/.../NumberUtils.java
+++ ./src/main/.../NumberUtils.java
@@ -449,9 +449,7 @@
         if (StringUtils.isBlank(str)) {
             throw new NumberFormatException("A blank string is not a valid
    number");
         }
-        if (str.startsWith("--")) {
-            return null;
-        }
+
         if (str.startsWith("0x") || str.startsWith("-0x") || str.startsWith("0X
    ") || str.startsWith("-0X")) {
```

Figure 6.9: A correct generated patch for Defects4J Lang-7.

[5] https://issues.apache.org/jira/browse/LANG-822

BATS assumes that similar buggy programs require similar patches to fix. To predict the generated patch correctness, BATS searches for a buggy program that fails on similar failing test cases with Lang-7. The retrieved program is the bug Lang-16 [6] in Defects4J. BATS predicts the generated patch is correct if it's similar with the developer patch addressing bug Lang-16. However, the retrieved bug Lang-16 is not related with bug Lang-7 even though they have similar test cases and require a dissimilar patch to fix. Thus, BATS fail to predict the generated patch correctness.

Consider however the NL description of the patch as it is generated by CodeTrans:

```
removed the unnecessary "" -- "" from NumberUtils . startsWith (
) , it was restricting our.
```

The syntactic and semantic correlation between the bug and patch description is obvious, which supports the fact that QUATRAIN predicts the patch as correct.

6.6.3 Threats to Validity

The implementation of QUATRAIN uses a pre-trained BERT to embed bug and patch descriptions before feeding them into the QA model. QUATRAIN also uses CodeTrans to generate patch descriptions. These choices may have influenced greatly our results. The associated threat is nevertheless limited since these constitute the state-of-the-art in their respective domains.

Our evaluation dataset includes 9,135 patches, though it is highly imbalanced (83% incorrect vs. 17% correct patches). This imbalance may bias our results. We mitigate this bias by stressing more on AUC metric, rather than F1 score and by performing comparison experiments against the state-of-the-art.

Our patch correctness labels have been manually decided in prior work [4]. The accuracy of the labels and the ground truth constitute a threat to validity [241], which is mitigated in part by our comparison against the state-of-the-art on the same datasets.

Our experimental evaluation does not perform any fine-tuning of the hyper-parameters of the QA model or even the initial BERT model used for embedding bug and patch descriptions. The yielded performance may thus not be representative of what can be achieved.

[6] An upper-case hex bug in https://issues.apache.org/jira/browse/LANG-746

6.7 Conclusion

In this paper, we present a novel perspective to the patch correctness assessment problem in automated program repair. Given a plausible patch, which is validated by an imperfect oracle, the need for correctness identification is acute, as several studies have revealed that state-of-the-art repair tools generate overfitting patches. Our idea is that a correct patch is the one that answers to the problem revealed by the execution failure (bug). We therefore design Quatrain, a neural network architecture that leverages NLP to learn to correlate bugs and patch descriptions and produce a Question-Answering based classifier. Given a buggy program, we consider its bug report and leverage CodeTrans to generate descriptions for all APR-generated patches targeting the bug. Then, we use these NL descriptions of bugs and patches to feed the QA classifier of Quatrain. The classification decision serves as a prediction of patch correctness. The experimental results show that our approach identifies 92.7% correct patches and filter 61.7% incorrect patches with an AUC of 0.886. We then investigate and discuss the influence of the quality of the input (bug report and code change description) on the effectiveness of Quatrain. We also perform experiments to demonstrate that Quatrain indeed learns and builds on the correlation between the bug report and the patch to make the predictions. Finally, we reproduce recent state-of-the-art static and dynamic patch assessment tools on our dataset and show that Quatrain exhibits comparable or better effectiveness in recalling correct patches and filtering out incorrect patches. Insights from our work open new research directions in patch assessment, but also provide a novel use case for a large body of the literature that is focused on commit message generation. Additionally, we underscore the importance of high-quality bug reports to the APR community. Moving forward, a promising research direction for assessing patch correctness involves continuing to establish connections between the semantics of bugs and code changes.

7 Conclusion

In this dissertation, we presented four research focuses to address the patch overfitting problem that threatens the practical adoption of APR techniques. Given the limitations of existing approaches and challenges, our work is devoted to assessing patch correctness by statically capturing the semantics of a correct patch. Specifically, we divide the work into two parts: 1) evaluating representation learning of code changes and 2) identifying the semantic correlation between a correct patch and the bug.

The objective of the first part is to investigate whether we can capture patch semantics for patch correctness by evaluating representation learning of code changes. By leveraging advances in representation learning models (*e.g.,* LLMs), we first conducted an empirical study of a widely-spread hypothesis in filtering out incorrect patches. Then, we built a patch correctness prediction framework Leopard with the embedding models and machine learning algorithms. Furthermore, since engineered features crafted by humans have semantics, we proposed combining both features to improve the performance. Towards accurate prediction, we developed Panther, the upgraded version of Leopard, that combines learned and engineered features in three methods. Experimental results show that Panther outperforms state of the art approaches. When applying the XGBoost classifier with BERT embeddings, Panther achieves the highest score with the metrics: an AUC value of 0.82 and an F-Measure score of 0.79. Finally, we conducted an analysis using an explainable ML technique SHAP on features and classifiers to enhance the understanding of the essence of identifying patch correctness and inspire the development of future approaches.

In the second part, we presented two approaches that identify the behavior of a correct patch in relation to the bug. We first heuristically proposed that different buggy programs may require similar code changes if they both fail on similar test cases. We then validated the hypothesis by performing hierarchical clustering based on the semantic embeddings of test cases and patches. Building upon this hypothesis, we introduced BATS (Behaviour Against failing Test Specification), an unsupervised approach to predict patch correctness by statically checking the behavioral similarity of generated patches against historical projects' correct patches that correspond to failing test cases that are similar to the failing tests of the bug is resolved. To evaluate the effectiveness of BATS, we collected a large dataset of plausible patches generated by 32 APR tools or extracted from defects benchmarks. We applied BATS to this dataset and measured its performance in accurately identifying correct patches while filtering out incorrect ones. Furthermore, we proposed another novel perspective by formulating the problem of patch correctness assessment as a question-answering task. To address this challenge, we presented Quatrain, a supervised learning approach that leverages a deep NLP model to classify the semantic correlation between a

bug (question) and a patch (answer). In our evaluation, we compared QUATRAIN against state-of-the-art static and dynamic approaches using three commonly used benchmarks. The results demonstrated that QUATRAIN achieves comparable or better performance in patch correctness assessment, as measured by metrics such as AUC, F1, +Recall, and -Recall.

In summary, this dissertation contributed to the field of APR by introducing static assessment approaches to assess the correctness of patches produced by APR tools. In particular, we have identified the key of patch correctness assessment: identifying the semantic correlation between a correct patch and its associated bug. Our methodologies enhanced the comprehensiveness, soundness, and precision of static approaches for evaluating patch correctness, paving the way for future research aimed at learning the behavior of correct patches.

8 Future Work

In this chapter, we present potential future research directions that are in line with this dissertation.

Contents

8.1 Learning to Represent Patches

A software patch, which represents the source code differences between two software versions, is the primary outcome of the APR process. In recent years, building upon empirical insights concerning the repetitiveness of code changes [42], numerous approaches have utilized machine learning models based on patch datasets to learn the deep representation of the patch. This has facilitated the automation of various tasks, such as just-in-time defect prediction [242, 243], code completion [244, 245, 246, 247, 156], and patch correctness assessment [81, 4]. State-of-the-art patch representation approaches have endeavoured to capture the inherent semantic and structural information in source code through token and AST data [248, 249, 250]. Specifically, this dissertation leverages advanced representation learning to evaluate patch correctness across our four research focuses. Unfortunately, these methods do not explicitly represent the code differences but rather depend on representing the code before and after the change, although adding some ad-hoc annotations to highlight the changes for the trained model. Therefore, a promising direction for future research could involve integrating contextual information with AST-based and token-based differences to improve patch representation learning and to more effectively identify the nuances of a given patch in the field of patch correctness assessment.

8.2 Capturing the Semantics of the Bug

To accurately assess patch correctness, it is crucial to grasp the underlying semantics of the bug in question. In this dissertation, we introduce BATS, a tool designed to compare patch behaviors by measuring the similarity between the functions of failing test cases associated with the bug. However, this approach has limitations, as it does not take into account dynamic execution information triggered by the bug. On the other hand, Quatrain employs bug reports to represent the semantics of the bug. Unfortunately, the quality of such reports is often suboptimal, leading to a loss of valuable semantic information. Additionally, our methodologies neglect to consider the original source code containing the bug itself. Recent research has shown that ChatGPT can effectively discern the intentions behind buggy programs and identify failure-inducing test cases [251]. This capability offers a valuable avenue for capturing more nuances of the semantics of a bug. Consequently, we believe a promising avenue for enhancing patch correctness assessment lies in utilizing LLMs to extract a more comprehensive understanding of the semantics of a bug and correlate this understanding with the behavior of correct patches.

8.3 Integrating Patch Correctness Assessment with Heuristic-based APR

Heuristic-based APR usually involves the construction of a vast search space for generating patches. As such, the efficient identification of correct patches becomes a key objective of these techniques, especially when dealing with the challenges posed by patch overfitting [25]. In response, state of the art strategies proposed constraining the search space to generate or prioritize high-quality patches [46, 41]. However, these methodologies tend to depend on the extraction of dynamic execution information, a process that can be time-consuming and restrict search efficiency.

Moreover, the correct behavior of the patch corresponds to the bug it addresses, which has not been identified during the patch generation. Meanwhile, static patch correctness assessment approaches have gained attention due to their effectiveness and efficiency [4, 81]. Nonetheless, these techniques are often perceived as a separate phase, independent from patch generation, and consequently do not facilitate the exploration of the patch space. Given this, investigating the integration of static patch correctness assessment strategies with heuristic-based APR is a worthwhile endeavor in future research. Such integration could potentially enhance the efficiency of the search process for correct patches.

8.4 Overfitting in LLMs-based Repair

The recent emergence of large-scale language models (LLMs) has received much attention in society in general. LLMs, which are pre-trained on vast amounts of source code and natural language data, have demonstrated exceptional capabilities in understanding code structures and generating codes or texts. Advances led by LLMs have further enhanced the effectiveness of automatic techniques for program repair by generating more correct patches than the state of the arts [252, 253, 254, 255, 69, 70, 256]. However, existing program repair studies with LLMs tend to assess their performance using old publicly available benchmark data, which may have been leaked into the training corpus of the LLMs. This experimental bias potentially jeopardizes the generalizability of reported results to new and unseen problems. In particular, our recent study preliminarily validated the generalizability problem of the LLMs and found that the responses generated by LLMs may overfit to their training corpus [68]. Therefore, the evaluation or identification of overfitting in LLM-based repair represents a promising research direction warranting further investigation.

Research Activities

We finally present the research activities undertaken during my Ph.D. journey. Specifically, we list: 1) the papers to which we contributed; 2) the tools and datasets produced through our research; 3) the venues where I served.

List of Papers

Papers included in this dissertation:

- [**ASE'20**] **Haoye Tian**, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 981-992. 2020.
- [**TOSEM'22, ICSE'23 Journal First**] **Haoye Tian**, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, no. 4 (2022): 1-30.
- [**ASE'22**] **Haoye Tian**, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and TegawendÉ F. BissyandÉ. Is this Change the Answer to that Problem? Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1-13. 2022.
- [**TOSEM'23**] **Haoye Tian**, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F. Bissyandé. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. *ACM Transactions on Software Engineering and Methodology* 32, no. 4 (2023): 1-34.

Papers not included in this dissertation:

- [**Under Review**] **Haoye Tian**, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. Is ChatGPT the Ultimate Programming Assistant–How far is it? arXiv preprint arXiv:2304.11938 (2023).
- [**EMSE'21**] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, **Haoye Tian**, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F. Bissyandé. Where were the repair ingredients for Defects4j bugs? Exploring the impact of repair ingredient retrieval on the performance of 24 program

repair systems. *Empirical Software Engineering* 26 (2021): 1-33.

- [**AAAI'23**] Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, **Haoye Tian**, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. MetaTPTrans: A Meta Learning Approach for Multilingual Code Representation Learning. The *37th AAAI Conference on Artificial Intelligence*, 2023.
- [**ASE'23**] Tsz-On Li, Wenxi Zong, Yibo Wang, **Haoye Tian**, Ying Wang, and Shing-Chi Cheung. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *38th IEEE/ACM International Conference on Automated Software Engineering*, 2023.

Tools and Datasets

- **LEOPARD:** https://github.com/SerVal-DTF/DL4PatchCorrectness
- **PANTHER:** https://github.com/HaoyeTianCoder/Panther
- **BATS:** https://github.com/HaoyeTianCoder/BATS
- **QUATRAIN:** https://github.com/Trustworthy-Software/Quatrain
- **ChatGPT-Study:** https://github.com/HaoyeTianCoder/ChatGPT-Study

Services

- **Organizing Committee**: ASE'23 Publicity Co-Chair
- **Program Committee:** SANER'24 Tools Demo PC, A-Mobile'23 PC, ESEC/FSE'23 AE PC, ISSTA'23 AE PC, MSR'23 Junior PC
- **Journal Referee:** TSE'23, EMSE'23, TSE'22, STVR'21, JCST'21
- **Conference Reviewer:** ISSTA'23, ICSE'22, ICSE'21, ICSE'20 , ASE'20

Bibliography

[1] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system," *Empirical Software Engineering*, vol. 24, pp. 33–67, 2019.

[2] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th international conference on software engineering*, pp. 789–799, 2018.

[3] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 981–992, IEEE, 2020.

[4] H. Tian, Y. Li, W. Pian, A. K. Kabore, K. Liu, A. Habib, J. Klein, and T. F. Bissyandé, "Predicting patch correctness based on the similarity of failing test cases," *ACM Transactions on Software Engineering and Methodology*, 2022.

[5] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th international conference on Software engineering*, pp. 33–42, 2006.

[6] C. Jones, *Applied software measurement*. McGraw-Hill Education, 2008.

[7] Wikipedia, "Knight capital group," 2023. Accessed: 2023-04-27.

[8] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, vol. 229, 2013.

[9] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[10] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[11] X. Gao, Y. Noller, and A. Roychoudhury, "Program repair," *arXiv preprint arXiv:2211.12787*, 2022.

[12] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference on artificial intelligence*, 2017.

[13] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1–12, IEEE, 2019.

[14] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 602–614, 2020.

[15] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 1506–1518, 2022.

[16] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1, pp. 213–224, IEEE, 2016.

[17] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2018.

[18] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Engineering*, vol. 20, pp. 176–205, 2015.

[19] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, pp. 1980–2024, 2020.

[20] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 772–781, IEEE, 2013.

[21] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "Jfix: semantics-based repair of java programs via symbolic pathfinder," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 376–379, 2017.

[22] S. Mechtaev, A. Griggio, A. Cimatti, and A. Roychoudhury, "Symbolic execution with existential second-order constraints," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 389–399, 2018.

[23] G. Yang, S. Khurshid, and M. Kim, "Specification-based test repair using a lightweight formal method," in *FM 2012: Formal Methods: 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings 18*, pp. 455–470, Springer, 2012.

[24] A. Nilizadeh, "Automated program repair and test overfitting: measurements and approaches using formal methods," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 480–482, IEEE, 2022.

[25] Z. Qi, F. Long, S. Achour, and M. Rinard, "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*, pp. 24–36, 2015.

[26] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 532–543, 2015.

[27] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2920–2938, 2021.

[28] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 302–313, 2019.

[29] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, 2021.

[30] D. Yang, K. Liu, D. Kim, A. Koyuncu, K. Kim, H. Tian, Y. Lei, X. Mao, J. Klein, and T. F. Bissyandé, "Where were the repair ingredients for defects4j bugs? exploring the impact of repair ingredient retrieval on the performance of 24 program repair systems," *Empirical Software Engineering*, vol. 26, pp. 1–33, 2021.

[31] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust enhancement issues in program repair," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2228–2240, 2022.

[32] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 524–535, IEEE, 2019.

[33] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 968–980, 2020.

[34] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp. 831–841, 2017.

[35] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, pp. 226–236, 2017.

[36] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 8–18, 2019.

[37] M. Böhme, C. Geethal, and V.-T. Pham, "Human-in-the-loop automatic program repair," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 274–285, IEEE, 2020.

[38] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 298–312, 2016.

[39] X. Gao and A. Roychoudhury, "Interactive patch generation and suggestion," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 17–18, 2020.

[40] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 727–738, 2016.

[41] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax-and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 593–604, 2017.

[42] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 306–317, ACM, 2014.

[43] J. Chen, A. F. Donaldson, A. Zeller, and H. Zhang, "Testing and verification of compilers (dagstuhl seminar 17502)," *Dagstuhl Reports*, vol. 7, no. 12, pp. 50–65, 2017.

[44] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*, pp. 416–426, IEEE, 2017.

[45] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pp. 255–266, IEEE, 2019.

[46] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 1–11, ACM, 2018.

[47] P. Cashin, C. Martinez, W. Weimer, and S. Forrest, "Understanding automatically-generated patches through symbolic invariant differences," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 411–414, IEEE, 2019.

[48] B. Yang and J. Yang, "Exploring the differences between plausible and correct patches at fine-grained level," in *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pp. 1–8, IEEE, 2020.

[49] S. Kang and S. Yoo, "Language models can prioritize patches for practical program patching," in *Proceedings of the Third International Workshop on Automated Program Repair*, pp. 8–15, 2022.

[50] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 31–42, 2019.

[51] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[52] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 254–265, 2014.

[53] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 166–178, 2015.

[54] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 802–811, IEEE, 2013.

[55] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. S. Păsăreanu, "Nn repair: constraint-based repair of neural network classifiers," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, pp. 3–25, Springer, 2021.

[56] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 390–405, 2021.

[57] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[58] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, pp. 691–701, 2016.

[59] T. Durieux and M. Monperrus, "Dynamoth: dynamic code synthesis for automatic program repair," in *Proceedings of the 11th International Workshop on Automation of Software Test*, pp. 85–91, 2016.

[60] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.

[61] OpenAI, "Chatgpt: Optimizing language models for dialogue," 2022.

[62] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, pp. 69–75, 2022.

[63] J. A. Prenner and R. Robbes, "Automatic program repair with openai's codex: Evaluating quixbugs," *arXiv preprint arXiv:2111.03922*, 2021.

[64] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2020.

[65] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[66] OpenAI, "Openai documentation," 2022.

[67] Github, "Copilot."

[68] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, "Is chatgpt the ultimate programming assistant–how far is it?," *arXiv preprint arXiv:2304.11938*, 2023.

[69] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," *arXiv preprint arXiv:2301.08653*, 2023.

[70] C. S. Xia and L. Zhang, "Conversational automated program repair," *arXiv preprint arXiv:2301.13246*, 2023.

[71] J. Jiang and Y. Xiong, "Can defects be fixed with weak test suites? an analysis of 50 defects from defects4j," *arXiv preprint arXiv:1705.04149*, 2017.

[72] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 55–64, 2010.

[73] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, "Quality of automated program repair on real-world defects," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 637–661, 2020.

[74] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 3–13, IEEE, 2012.

[75] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 356–366, IEEE, 2013.

[76] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *2013 IEEE International Conference on Software Maintenance*, pp. 180–189, IEEE, 2013.

[77] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 702–713, IEEE, 2016.

[78] X.-B. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 163–163, 2018.

[79] A. Nilizadeh, G. T. Leavens, X.-B. D. Le, C. S. Păsăreanu, and D. R. Cok, "Exploring true test overfitting in dynamic automated program repair using formal methods," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 229–240, IEEE, 2021.

[80] X. Kong, L. Zhang, W. E. Wong, and B. Li, "The impacts of techniques, programs and tests on automated program repair: An empirical study," *Journal of Systems and Software*, vol. 137, pp. 480–496, 2018.

[81] H. Tian, X. Tang, A. Habib, S. Wang, K. Liu, X. Xia, J. Klein, and T. F. Bissyandé, "Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–13, 2022.

[82] M. Motwani, "High-quality automated program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 309–314, IEEE, 2021.

[83] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[84] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury, *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 180–182, IEEE, 2017.

[85] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, "How different is it between machine-generated and developer-provided patches?: An empirical study on the correct patches generated by automated program repair techniques," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–12, IEEE, 2019.

[86] G. Bennett, T. Hall, and D. Bowes, "Some automatically generated patches are more likely to be correct than others: an analysis of defects4j patch features," in *Proceedings of the Third International Workshop on Automated Program Repair*, pp. 46–52, 2022.

[87] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st international conference on Software engineering*, pp. 213–224, 1999.

[88] R. Just, D. Jalali, and M. D. Ernst, "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*, pp. 437–440, 2014.

[89] L. Zemín, S. G. Brida, A. Godio, C. Cornejo, R. Degiovanni, G. Regis, N. Aguirre, and M. Frias, "An analysis of the suitability of test-based patch acceptance criteria," in *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, pp. 14–20, IEEE, 2017.

[90] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[91] J. Jiang, Y. Xiong, and X. Xia, "A manual inspection of defects4j bugs and its implications for automatic program repair," *Science china information sciences*, vol. 62, no. 10, pp. 1–16, 2019.

[92] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pp. 55–56, 2017.

[93] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury, "A correlation study between automated program repair and test-suite metrics," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 24–24, 2018.

[94] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," *Journal of Systems and Software*, vol. 171, p. 110825, 2021.

[95] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE transactions on Software Engineering*, vol. 38, no. 2, pp. 258–277, 2011.

[96] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *Ieee transactions on software engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[97] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "An empirical analysis of the influence of fault space on search-based automated program repair," *arXiv preprint arXiv:1707.05172*, 2017.

[98] D. Yang, Y. Qi, and X. Mao, "Evaluating the strategies of statement selection in automated program repair," in *Software Analysis, Testing, and Evolution: 8th International Conference, SATE 2018, Shenzhen, Guangdong, China, November 23–24, 2018, Proceedings 8*, pp. 33–48, Springer, 2018.

[99] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. Le Goues, "Sosrepair: Expressive semantic search for real-world program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2162–2181, 2019.

[100] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE conference on software testing, validation and verification (ICST)*, pp. 102–113, IEEE, 2019.

[101] F. Y. Assiri and J. M. Bieman, "Fault localization for automated program repair: effectiveness, performance, repair correctness," *Software Quality Journal*, vol. 25, pp. 171–199, 2017.

[102] X. Xu, Y. Sui, H. Yan, and J. Xue, "Vfix: value-flow-guided precise program repair for null pointer dereferences," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 512–523, IEEE, 2019.

[103] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 448–458, IEEE, 2015.

[104] T. Xu, L. Chen, Y. Pei, T. Zhang, M. Pan, and C. A. Furia, "Restore: Retrospective fault localization enhancing automated program repair," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 309–326, 2020.

[105] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 511–526, 2005.

[106] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!," in *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 67–76, 2008.

[107] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "ifixr: Bug report driven program repair," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 314–325, 2019.

[108] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 637–647, IEEE, 2017.

[109] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts: An extended study," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2841–2857, 2020.

[110] M. Motwani and Y. Brun, "Better automatic program repair by using bug reports and tests together," in *International Conference on Software Engineering (ICSE)*, 2023.

[111] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 295–306, IEEE, 2015.

[112] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 151–162, 2018.

[113] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 19–30, 2019.

[114] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 471–482, IEEE, 2015.

[115] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 129–139, 2018.

[116] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pp. 383–401, Springer, 2016.

[117] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–27, 2021.

[118] G. C. Website, "Gnu coreutils.," 2022.

[119] B. Website, "Busybox," 2022.

[120] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 188–199, IEEE, 2019.

[121] M. P. Gissurarson, L. Applis, A. Panichella, A. van Deursen, and D. Sands, "Propr: property-based automatic program repair," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 1768–1780, 2022.

[122] Y. Yuan and W. Banzhaf, "A hybrid evolutionary system for automatic software repair," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1417–1425, 2019.

[123] A. Ghanbari, "Objsim: Lightweight automatic patch prioritization via object similarity," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 541–544, 2020.

[124] A. Ghanbari and A. Marcus, "Patch correctness assessment in automated program repair based on the impact of patches on production and test code," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 654–665, 2022.

[125] V. Csuvik, D. Horváth, F. Horváth, and L. Vidács, "Utilizing source code embeddings to identify correct patches," in *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pp. 18–25, IEEE, 2020.

[126] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–29, 2022.

[127] Q.-N. Phung, M. Kim, and E. Lee, "Identifying incorrect patches in program repair based on meaning of source code," *IEEE Access*, vol. 10, pp. 12012–12030, 2022.

[128] Y. Dong, D. Tang, X. Cheng, and Y. Yang, "Quality evaluation method of automatic software repair using syntax distance metrics," *Symmetry*, vol. 14, no. 8, p. 1751, 2022.

[129] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 660–670, IEEE, 2017.

[130] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th international conference on software engineering*, pp. 12–23, 2018.

[131] R. K. Saha, H. Yoshida, M. R. Prasad, S. Tokumoto, K. Takayama, and I. Nanba, "Elixir: an automated repair tool for java programs," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 77–80, 2018.

[132] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pp. 298–309, 2018.

[133] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 428–439, 2016.

[134] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 121–130, 2011.

[135] Y. Yuan and W. Banzhaf, "Toward better evolutionary program repair: An integrated approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 1, pp. 1–53, 2020.

[136] D. Yan, K. Liu, Y. Niu, L. Li, Z. Liu, Z. Liu, J. Klein, and T. F. Bissyandé, "Crex: Predicting patch correctness in automated repair of c programs through transfer learning of execution semantics," *Information and Software Technology*, vol. 152, p. 107043, 2022.

[137] M. Martinez, M. Kechagia, A. Perera, J. Petke, F. Sarro, and A. Aleti, "Test-based patch clustering for automatically-generated patches assessment," *arXiv preprint arXiv:2207.11082*, 2022.

[138] S. P. Reiss and Q. Xin, "A quick repair facility for debugging," *arXiv preprint arXiv:2202.05577*, 2022.

[139] J. Lee, S. Hong, and H. Oh, "Npex: repairing java null pointer exceptions without tests," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 1532–1544, 2022.

[140] E. B. Bae, "Effective and efficient patch validation via differential fuzzing," 2022.

[141] H. Cao, F. Liu, J. Shi, Y. Chu, and M. Deng, "Automated repair of java programs with random search via code similarity," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 470–477, IEEE, 2021.

[142] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[143] M. Monperrus, "The living review on automated program repair," in *HAL/archives-ouvertes. fr, Technical Report*, 2018.

[144] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 364–374, IEEE, 2009.

[145] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 456–467, IEEE, 2019.

[146] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proceedings of the 41st International Conference on Software Engineering*, pp. 13–24, IEEE, 2019.

[147] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "LSRepair: Live search of fix ingredients for automated program repair," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference ERA Track*, pp. 658–662, IEEE, 2018.

[148] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, 2021.

[149] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," in *Proceedings of the 1st International Workshop on Intelligent Bug Fixing*, pp. 1–10, IEEE, 2019.

[150] M. Soto and C. Le Goues, "Using a probabilistic model to predict bug fixes," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 221–231, IEEE, 2018.

[151] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 297–308, ACM, 2016.

[152] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 281–293, ACM, 2014.

[153] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[154] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "CodeBERT: a pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[155] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4171–4186, 2019.

[156] W. Pian, H. Peng, X. Tang, T. Sun, H. Tian, A. Habib, J. Klein, and T. F. Bissyandé, "Metatptrans: A meta learning approach for multilingual code representation learning," *arXiv preprint arXiv:2206.06460*, 2022.

[157] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 141–151, 2018.

[158] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, p. 516–527, ACM, 2020.

[159] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on Machine Learning*, pp. 1188–1196, JMLR.org, 2014.

[160] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pp. 3034–3040, Morgan Kaufmann, 2017.

[161] S. Ndichu, S. Kim, S. Ozawa, T. Misu, and K. Makishima, "A machine learning approach to detection of javascript-based attacks using AST features and paragraph vectors," *Applied Soft Computing*, vol. 84, 2019.

[162] S. Zhou, B. Shen, and H. Zhong, "Lancer: Your code tell me what you need," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1202–1205, IEEE, 2019.

[163] T. Hoang, H. J. Kang, J. Lawall, and D. Lo, "CC2Vec: distributed representations of code changes," in *Proceedings of the 42nd International Conference on Software Engineering*, pp. 518–529, ACM, 2020.

[164] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 837–847, IEEE, 2012.

[165] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. A. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, no. 4, pp. 81:1–81:37, 2018.

[166] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, 2018.

[167] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[168] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[169] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 1145–1152, AAAI, 2020.

[170] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Proceedings of the 17th Mining Software Repositories*, ACM, 2020.

[171] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon, "A closer look at real-world patches," in *Proceedings of the 34th International Conference on Software Maintenance and Evolution*, pp. 275–286, IEEE, 2018.

[172] R. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*, pp. 10–13, ACM, 2018.

[173] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "BEARS: an extensible java bug benchmark for automatic program repair studies," in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, pp. 468–478, IEEE, 2019.

[174] R. Karampatsis and C. A. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the 17th Mining Software Repositories*, IEEE, 2020.

[175] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the 42nd International Conference on Software Engineering*, pp. 625–627, ACM, 2020.

[176] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[177] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[178] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "PatchNet: hierarchical deep learning-based stable patch identification for the linux kernel," *CoRR*, vol. abs/1911.03576, 2019.

[179] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 159:1–159:27, 2019.

[180] F. Madeiral, T. Durieux, V. Sobreira, and M. Maia, "Towards an automated approach for bug fix pattern detection," 2018.

[181] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 4765–4774, Curran Associates, Inc., 2017.

[182] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *Empirical Software Engineering*, vol. 26, no. 2, pp. 1–38, 2021.

[183] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*, pp. 7–10, 2016.

[184] T. G. Dietterich, "Approximate statistical tests for comparing supervised classification learning algorithms," *Neural computation*, vol. 10, no. 7, pp. 1895–1923, 1998.

[185] F. Tsimpourlas, A. Rajan, and M. Allamanis, "Learning to encode and classify test executions," *arXiv preprint arXiv:2001.02444*, 2020.

[186] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[187] Q. Huang, A. Qiu, M. Zhong, and Y. Wang, "A code-description representation learning model based on attention," in *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, pp. 447–455, IEEE, 2020.

[188] M. Boehme, *Automated regression testing and verification of complex code changes*. PhD thesis, National University of Singapore, 2014.

[189] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, "Automatic repair of real bugs: An experience report on the defects4j dataset," *CoRR*, vol. abs/1505.07002, 2015.

[190] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," *arXiv preprint arXiv:1811.05703*, 2018.

[191] M. Martinez and M. Monperrus, "ASTOR: a program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 441–444, ACM, 2016.

[192] Y. Yuan and W. Banzhaf, "ARJA: automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, 2020.

[193] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: the cardumen mode of astor," in *Proceedings of the 10th International Symposium on Search Based Software Engineering*, pp. 65–86, Springer, 2018.

[194] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 101–114, ACM, 2020.

[195] J. Kim and S. Kim, "Automatic patch generation with context-based change application," *Empirical Software Engineering*, vol. 24, no. 6, pp. 4071–4106, 2019.

[196] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*, pp. 479–490, IEEE, 2019.

[197] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: effective object-oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 648–659, IEEE, 2017.

[198] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," tech. rep., Stanford, 2006.

[199] M. S. G. Karypis, V. Kumar, and M. Steinbach, "A comparison of document clustering techniques," in *TextMining Workshop at KDD2000 (May 2000)*, 2000.

[200] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, "Opportunities and challenges in code search tools," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[201] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 130–140, IEEE, 2018.

[202] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2021.

[203] C. Peng and A. Rajan, "Automated test generation for opencl kernels using fuzzing and constraint solving," in *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pp. 61–70, 2020.

[204] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller, "Probabilistic grammar-based test generation," *Software Engineering 2021*, vol. P-310, pp. 97–98, 2021.

[205] M. Selakovic, M. Pradel, R. Karim, and F. Tip, "Test generation for higher-order functions in dynamic languages," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[206] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "FaCoY: a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 946–957, ACM, 2018.

[207] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "CLCDSA: Cross language code clone detection using syntactical features and api documentation," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1026–1037, 2019.

[208] G. Mathew, C. Parnin, and K. T. Stolee, "SLACC: Simion-based language agnostic code clones," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 210–221, 2020.

[209] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, 2021.

[210] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 480–491, 2016.

[211] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 363–376, 2017.

[212] J. Gao, X. Yang, Y. Fu, Y. Jiang, H. Shi, and J. Sun, "VulSeeker-pro: enhanced semantic learning based binary vulnerability seeker with emulation," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 803–808, 2018.

[213] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

[214] M. Monperrus, "The living review on automated program repair," 2020.

[215] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," 2022.

[216] H. Tian, K. Liu, Y. Li, A. K. Kaboré, A. Koyuncu, A. Habib, L. Li, J. Wen, J. Klein, and T. F. Bissyandé, "The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches," *arXiv preprint arXiv:2203.08912*, 2022.

[217] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[218] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1161–1173, IEEE, 2021.

[219] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 505–509, IEEE, 2021.

[220] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 308–318, 2008.

[221] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *2013 IEEE Sixth international conference on software testing, verification and validation*, pp. 282–291, IEEE, 2013.

[222] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 141–152, 2018.

[223] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond, "Recdroid: automatically reproducing android application crashes from bug reports," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 128–139, IEEE, 2019.

[224] A. Khanfir, A. Koyuncu, M. Papadakis, M. Cordy, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Ibir: Bug report driven fault injection," *arXiv preprint arXiv:2012.06506*, 2020.

[225] M. Kim, Y. Kim, and E. Lee, "Denchmark: A bug benchmark of deep learning-related software," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 540–544, IEEE, 2021.

[226] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[227] A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes, and B. Rost, "Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing," *arXiv preprint arXiv:2104.02443*, 2021.

[228] P. Bafna, D. Pramod, and A. Vaidya, "Document clustering: Tf-idf approach," in *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, pp. 61–66, IEEE, 2016.

[229] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[230] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[231] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.

[232] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[233] S. Min, M. Seo, and H. Hajishirzi, "Question answering through transfer learning from large fine-grained supervision data," *arXiv preprint arXiv:1702.02171*, 2017.

[234] Z. Gao, X. Xia, D. Lo, J. Grundy, and Y.-F. Li, "Code2que: A tool for improving question titles from mined code snippets in stack overflow," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1525–1529, 2021.

[235] M. Tan, C. d. Santos, B. Xiang, and B. Zhou, "Lstm-based deep learning models for non-factoid answer selection," *arXiv preprint arXiv:1511.04108*, 2015.

[236] M. Hossin and M. N. Sulaiman, "A review on evaluation metrics for data classification evaluations," *International journal of data mining & knowledge management process*, vol. 5, no. 2, p. 1, 2015.

[237] L. A. Jeni, J. F. Cohn, and F. De La Torre, "Facing imbalanced data–recommendations for the use of performance metrics," in *2013 Humaine association conference on affective computing and intelligent interaction*, pp. 245–251, IEEE, 2013.

[238] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 143–153, 2019.

[239] P. Irolla and A. Dey, "The duplication issue within the drebin dataset," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 3, pp. 245–249, 2018.

[240] Y. Zhao, L. Li, H. Wang, H. Cai, T. F. Bissyandé, J. Klein, and J. Grundy, "On the impact of sample duplication in machine-learning-based android malware detection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–38, 2021.

[241] D. Yang, Y. Lei, X. Mao, D. Lo, H. Xie, and M. Yan, "Is the ground truth really accurate? dataset purification for automated program repair," in *2021 IEEE 28th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2021.

[242] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 34–45, IEEE, 2019.

[243] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[244] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2727–2735, 2019.

[245] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 473–485, 2020.

[246] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *Proceedings of the 28th International Conference on Program Comprehension*, pp. 37–47, 2020.

[247] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of bert models for code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 108–119, IEEE, 2021.

[248] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794, IEEE, 2019.

[249] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *PACMPL*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[250] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao, "Fira: Fine-grained graph-based code change representation for automated commit message generation," 2022.

[251] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, and S.-C. Cheung, "Finding failure-inducing test cases with chatgpt," 2023.

[252] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.

[253] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," *arXiv preprint arXiv:2210.14179*, 2022.

[254] J. Zhang, J. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen, "Repairing bugs in python assignments using large language models," *arXiv preprint arXiv:2209.14876*, 2022.

[255] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *arXiv preprint arXiv:2302.05020*, 2023.

[256] N. M. S. Surameery and M. Y. Shakor, "Use chat gpt to solve programming bugs," *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, vol. 3, no. 01, pp. 17–22, 2023.