九州大学学術情報リポジトリ
Kyushu University Institutional Repository

Debugging of AI-enabled and Quantum Software Systems

石本, 優太

https://hdl.handle.net/2324/7443462

出版情報：Kyushu University, 2025, 博士（情報科学）, 課程博士
バージョン：
権利関係：

KYUSHU UNIVERSITY

Debugging AI-enabled and Quantum Software Systems

(AIおよび量子ソフトウェアシステムに対するデバッギング)

by

Yuta Ishimoto

石本 優太

A thesis submitted to the
Graduate School of Information Science and Electrical Engineering
in conformity with the requirements for
the degree of Doctor of Information Science



九州大学
KYUSHU UNIVERSITY

Kyushu University
Fukuoka, Japan
January 2026

Abstract

In recent years, software systems enhanced with the capabilities of artificial intelligence (AI) and quantum computation have attracted increasing attention. We refer to these systems as *AI-enabled software systems* and *quantum software systems*, respectively. These systems enable tasks that were difficult or even impossible to accomplish with conventional software. For example, AI-enabled software systems include applications such as autonomous driving and chatbots through inductive development paradigms that learn from large amounts of data. On the other hand, quantum software systems are expected to enhance applications such as optimization and cryptography by leveraging their computational power. Given their increasing deployment in critical domains, misbehavior in these systems can lead to threats to human life or substantial economic losses. This underscores the need for rigorous software engineering practices for these software systems, just as for conventional software systems.

Among software engineering practices, *debugging*, the process of isolating and correcting mistakes in computer software, is essential for ensuring software quality. To support this process, the software engineering research community has dedicated considerable effort, including automated fault localization and automated program repair. However, conventional debugging techniques may not be well suited to emerging software paradigms such as AI-enabled and quantum software systems. For AI-enabled software systems, debugging requires attention not only to the source code but also to the datasets and trained AI models, as their behavior depends on data-driven learning processes. For quantum software systems, debugging should take into account the quantum mechanical properties of quantum software, such as superposition and entanglement of quantum bits (qubits). These challenges motivate the development of novel debugging methodologies tailored to AI-enabled and quantum software systems.

This dissertation addresses the challenges of debugging AI-enabled and quantum software systems through four complementary studies related to fault localization and repair. The main contributions of this dissertation are as follows: the first two concern debugging for AI-enabled software systems, and the remaining two concern debugging for quantum software systems.

- *Probabilistic Automaton-based Fault Localization for Recurrent Neural Networks.* Deep neural networks (DNNs) play a central role in AI-enabled software systems. Among various types of DNNs, recurrent neural networks (RNNs) present unique debugging challenges because they maintain internal states that dynamically change depending on input sequences. To address this, we propose probabilistic automaton-based fault localization (PAFL). PAFL abstracts RNNs into probabilistic finite automata (PFAs) and localizes faulty sequences of state transitions on PFAs by adapting the concept of spectrum-based fault localization (SBFL). Our evaluation demonstrates that PAFL can identify data samples leading to misbehavior more effectively and efficiently than the state-of-the-art techniques in 66.74% of the cases.
- *Repairs and Breaks Prediction for Deep Neural Networks.* DNN repair methods refer to those that target components of the model itself (e.g., neurons or weights) to reduce the specific type of misbehavior while preserving its existing correct behavior. These methods are often computationally expensive. Therefore, predicting promising repair methods in advance is crucial to reduce the number of trial-and-error iterations during the debugging process of AI-enabled software systems. To this end, we build prediction models to estimate the outcomes of DNN repair methods. Our evaluation demonstrates that these prediction models can reduce the overall time for repair application scenarios, yielding an average time savings of 16.29% compared to the naive method of applying all repair methods and checking the results.
- *Evaluating Mutation-based Fault Localization for Quantum Programs.* Quantum programs, consist of sequences of operations on qubits, known as quantum gates. While existing studies have demonstrated the effectiveness of mutation operators (e.g., deleting quantum gates) for testing quantum programs, their potential for fault localization remains unexplored. Therefore, we evaluate the effectiveness of mutation-based fault localization (MBFL) for quantum programs. While MBFL was originally proposed for classical (non-quantum) programs, we first evaluate its effectiveness in the quantum context. The experimental results demonstrate that mutation operations specific to quantum programs are more effective than classical mutation operations.
- *Leveraging Mutation Analysis for LLM-based Repair of Quantum Programs.* Although automated program repair (APR) techniques for quantum programs ex-

its, they often show low success rates or produce patches with limited interpretability. To address these limitations, we enhance LLM-based repair for quantum programs by incorporating insights from mutation analysis. We hypothesize that mutation analysis results can provide richer dynamic context than standard execution results (e.g., simple pass/fail outcomes), thereby improving both repair success and explanation quality for repair. Our evaluation demonstrates that incorporating mutation analysis results into the prompt enabled the LLM to repair 17 out of 18 faulty programs. Furthermore, it improves the quality of the generated explanations.

In summary, this dissertation contributes to advancing debugging for AI-enabled and quantum software systems by proposing novel approaches and evaluating the effectiveness of existing techniques for fault localization and repair.

Acknowledgement

My PhD journey has been supported by many people, to whom I am deeply grateful. First, I would like to express my profound gratitude to Professor Yasutaka Kamei, my supervisor. His continuous support, thoughtful guidance, and generous encouragement made it possible for me to complete this research. I am also sincerely grateful to Professor Naoyasu Ubayashi, who supervised me from the time I joined the laboratory as an undergraduate student through my first year of the doctoral course. Even after he moved to Waseda University, he continued to give me valuable advice on my research. Since the very beginning of my research in software engineering, he has provided invaluable feedback and unwavering support. I would also like to express my deep appreciation to Assistant Professor Masanari Kondo. From my first English paper, he has served as a co-author and has given me numerous concrete and practical suggestions on academic writing and research methodology. I could not have completed this dissertation without the support of each of them.

I would also like to thank my dissertation committee members, Professor Jianjun Zhao, Associate Professor Lei Ma, Professor Hiroyuki Nakagawa, Professor Naoyasu Ubayashi, and Professor Yasutaka Kamei, for their valuable comments and feedback during the midterm presentations of my PhD research. Associate Professor Lei Ma kindly hosted my first research visit abroad in Alberta, Canada, and I am deeply grateful for his generous support and insightful discussions from the point of view of an expert in software engineering for AI, throughout and beyond the visit.

I would also like to thank Dr. Naoto Sato, Dr. Ryota Katsube, and Dr. Hideto Ogawa for their insightful feedback on my quantum software engineering research.

I would also like to express my gratitude to Associate Professor Fuyuki Ishikawa and Associate Professor Paolo Arcaini for hosting my one-month stay at National Institute of Informatics (NII), which provided me with a valuable opportunity to further deepen my research on DNN repair.

I would also like to thank Professor Masafumi Shimojo, who supervised me when I was a student at National Institute of Technology, Fukui College. He gave extensive feedback on my very first piece of academic writing and marked up my draft so thoroughly that it was almost red with comments. His guidance allowed me to experience, for the first time, the joy of doing

research and motivated me to continue my studies at the university.

Additionally, I would like to thank all the labmates I have worked with over the past six years for sharing their knowledge and expertise with me. They also helped me enjoy my time as a student, both inside and outside the lab. I would also like to thank Tomoko Inao and Miki Watanabe, our administrative staff members, for their support with my daily expense reports and travel procedures. I would also like to thank Terry Bogard and Kazuya Mishima, my favorite characters in *Super Smash Bros. Ultimate*. They provided a great way for me to relax and served as a fun interface for communicating with my labmates.

Finally, I would like to thank my family for their continuous love and support. I could not have come this far without their support, and I am deeply grateful to them.

Related Publications

This dissertation encompasses the findings in the following publications:

- (Chapter 3) **PAFL: Probabilistic Automaton-based Fault Localization for Recurrent Neural Networks.**
  Yuta Ishimoto, Masanari Kondo, Naoyasu Ubayashi, Yasutaka Kamei, *Information and Software Technology*, Volume 155, Article number: 107117, March 2023.

- (Chapter 4) **Repairs and Breaks Prediction for Deep Neural Networks.**
  Yuta Ishimoto, Masanari Kondo, Lei Ma, Naoyasu Ubayashi, Yasutaka Kamei, *ACM Transactions on Software Engineering and Methodology*, Volume 34, Issue 4, Article No.: 94, pp.1-42, May 2025.

- (Chapter 5) **Evaluating Mutation-based Fault Localization for Quantum Programs.**
  Yuta Ishimoto, Masanari Kondo, Naoyasu Ubayashi, Yasutaka Kamei, Ryota Katsube, Naoto Sato and Hideto Ogawa, In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, pp.666-671, June 2025.

The extended work section contains the results of the following study which is accepted for *Early Research Achievement Track of the 33rd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2026)*:

- (Chapter 6) **Leveraging Mutation Analysis for LLM-based Repair of Quantum Programs.**
  Chihiro Yoshida*, Yuta Ishimoto*†, Olivier Nourry, Masanari Kondo, Makoto Matsushita, Yasutaka Kamei, and Yoshiki Higo (∗: equal contribution, †: corresponding author).

The following publications are not included directly in this dissertation but were conducted in parallel with the research for this dissertation.

- **An Initial Analysis of Repair and Side-effect Prediction for Neural Networks.**
  Yuta Ishimoto, Ken Matsui, Masanari Kondo, Naoyasu Ubayashi, Yasutaka Kamei, In *Proceedings of the 2nd International Conference on AI Engineering – Software Engineering for AI*, pp.80-85, May 2023.

- **Uncertainty-aware Metamorphic Testing for Robust Object Detection Models.**
  Jianhong Wang, Yuta Ishimoto, Masanari Kondo, Yasutaka Kamei, Naoyasu Ubayashi, In *Proceedings of the 23rd International Conference on Software Quality, Reliability, and Security Companion*, pp.9-17, October 2023.

- **An Empirical Study on Self-Admitted Technical Debt in Quantum Software.**
  Yuta Ishimoto, Yuto Nakamura, Ryota Katsube, Naoto Sato, Hideto Ogawa, Masanari Kondo, Yasutaka Kamei, Naoyasu Ubayashi, In *Proceedings of the 31st Asia-Pacific Software Engineering Conference*, pp.41-50, December 2024.

- **How Far Have LLMs Come Toward Automated SATD Taxonomy Construction?**
  Sota Nakashima, Yuta Ishimoto, Masanari Kondo, Tao Xiao, Yasutaka Kamei, In *Proceedings of the 32nd Asia-Pacific Software Engineering Conference*, December 2025. [accepted, to appear]

Contents

Acknowledgement iv

Related Publications vi

1 Introduction 1

1 Motivation . . . 1
2 Dissertation Overview . . . 2
3 Dissertation Organization . . . 5

2 Related Work 6

1 Debugging Conventional Software . . . 6
1.1 Fault Localization for Conventional Software . . . 6
1.2 Repair for Conventional Software . . . 8
2 Debugging AI-enabled Software . . . 9
2.1 Fault Localization for AI-enabled Software . . . 9
2.2 Repair for AI-enabled Software . . . 11
3 Debugging Quantum Software . . . 12
3.1 Fault Localization for Quantum Software . . . 13
3.2 Repair for Quantum Software . . . 14

3 PAFL: Probabilistic Automaton-based Fault Localization for Recurrent Neural Networks 16

1 Introduction . . . 16
2 Challenges of Fault Localization for RNNs . . . 18
3 Our Approach . . . 19
3.1 Overview . . . 19
3.2 Phase 1: Approximate RNN to PFA . . . 20
3.3 Phase 2: Collect PFA Log . . . 22
3.4 Phase 3: Apply Fault Localization to PFA . . . 23

3.5 Phase 4: Select Data . . . 26
4 Experimental Setup . . . 27
4.1 Research Questions . . . 28
4.2 Overview . . . 28
4.3 Datasets . . . 29
4.4 Model Settings . . . 30
4.5 Evaluation Criteria . . . 31
4.6 Comparison with RNNRepair . . . 32
5 Experimental Results . . . 33
5.1 RQ1) Do ANS scores represent the misbehavior of RNNs? . . . 33
5.2 RQ2) Does PAFL identify data samples that cause the misbehavior of RNNs? 35
6 Discussion . . . 40
6.1 Is PAFL superior to SOTA FL in terms of execution time? . . . 40
6.2 Do PAFL parameters need to be optimized? . . . 42
6.3 How often do ties in ANS scores occur? . . . 44
6.4 Are PFAs superior to DFAs in our sample-based fault localization (PAFL)? 45
7 Threats to Validity . . . 47
7.1 Internal Validity . . . 47
7.2 External Validity . . . 48
8 Conclusion . . . 48

4 Repairs and Breaks Prediction for Deep Neural Networks 49
1 Introduction . . . 49
2 Motivation . . . 53
3 Methods . . . 54
3.1 Repairs and Breaks in Neural Network . . . 54
3.2 Repairs and Breaks Datasets . . . 56
3.3 Repairs and Breaks Prediction Models . . . 56
4 Experimental Subjects . . . 58
4.1 Datasets . . . 58
4.2 DNN Properties . . . 60
4.3 Neural Network Models . . . 63
4.4 Repair Methods . . . 64
5 Experimental Setup . . . 64
5.1 Research Questions . . . 64

5.2 Experimental Procedure . . . 65
5.3 Experimental Environment . . . 68
6 Results . . . 68
6.1 RQ1: (Differences) What are the characteristics of repaired and broken samples? . . . 68
6.2 RQ2: (Prediction performance) To what extent can repairs and breaks be predicted? . . . 70
6.3 RQ3: (Usefulness) Can repairs and breaks prediction models suggest a promising repair method in a shorter time compared to a naive method? . . . 74
6.4 RQ4: (Transferability) Are repairs and breaks prediction models transferable to other repair methods? . . . 78
6.5 RQ5: (Generality) Can the promising repair method selection be applied to DNN properties other than correctness? . . . 82
7 Discussion . . . 90
7.1 Use cases . . . 90
7.2 Limitations and Future Work . . . 91
7.3 Additional Analysis . . . 92
8 Threats to Validity . . . 97
8.1 Internal validity . . . 97
8.2 External validity . . . 98
8.3 Construct validity . . . 99
9 Conclusion . . . 99

5 Evaluating Mutation-based Fault Localization for Quantum Programs 100
1 Introduction . . . 100
2 Study Design . . . 101
2.1 Research Questions . . . 102
2.2 Bug Benchmarks . . . 102
2.3 Mutating Buggy Programs . . . 104
2.4 Suspiciousness Scores . . . 104
2.5 Evaluation Metrics for Fault Localization . . . 105
2.6 Experimental Environment . . . 105
3 Results . . . 106
3.1 RQ1: Artificial vs. Real-world Faults . . . 106
3.2 RQ2: Quantum vs. Classical Mutation Operations . . . 107

4 Discussion . . . 108
4.1 Why are real-world faults in quantum programs challenging for MBFL? . . . 108
4.2 Comparison between MBFL and SBFL . . . 109
5 Threats to Validity . . . 110
6 Conclusion . . . 111

6 Extended Work: Leveraging Mutation Analysis for LLM-based Repair of Quantum Programs 112
1 Introduction . . . 112
2 Study Design . . . 114
2.1 Bug Benchmark . . . 114
2.2 Prompt Construction . . . 116
2.3 LLM . . . 117
2.4 Evaluation Metrics . . . 117
3 Results . . . 118
3.1 RQ1: Can mutation analysis results improve the repair success rate of buggy quantum programs? . . . 118
3.2 RQ2: Can mutation analysis results improve the quality of explanations generated by LLMs? . . . 119
4 Discussion . . . 120
4.1 Unique Successful Repairs Across Prompt Configurations . . . 120
4.2 Future Research Plan . . . 121
5 Threats to Validity . . . 122
6 Conclusion . . . 122

7 Conclusion 123

Rerefences 125

List of Figures

1.1 Overview of this dissertation. . . . 3

2.1 Example faulty program and execution traces for three test cases. . . . 7
2.2 Differences between DNN and RNN. . . . 10
2.3 An example of a quantum program written in Qiskit (above) and corresponding quantum circuit (below). . . . 13

3.1 Overview of PAFL. . . . 20
3.2 An example of a structure of a PFA. . . . 21
3.3 Flow of hidden states abstraction ($k = 3$). . . . 22
3.4 The bigrams (i.e., n-grams of $n = 2$) of PFA states obtained by feeding a sample x_1. The parentheses after u_i represent the suspiciousness scores. . . . 23
3.5 Overview of our experiment. . . . 28
3.6 Average ANS scores for each n and dataset. . . . 34
3.7 Overview for RQ2 approach. . . . 36
3.8 Numbers of parameter settings in which PAFL-selected data lead to worse performance for each n. . . . 38
3.9 Parameters (k, n) at the lowest/highest in terms of accuracy in the selected data samples. . . . 43
3.10 Difficulty in predicting the data samples selected by PAFL with PFAs and PAFL with DFAs. . . . 46

4.1 The flow for construction and practical usage of repairs and breaks prediction models. . . . 51
4.2 Workflow to build repairs and breaks prediction models. . . . 55
4.3 The experimental procedure. . . . 66

4.4 Comparison of the four metrics between (a) repaired and non-repaired samples, and (b) broken and non-broken samples (correctness property). Each radar plot represents the average of the four metrics across all datasets. 71
4.5 Prediction performance of (a) repairs and (b) breaks prediction models for each repair method and dataset. 73
4.6 The flow of the experiment for RQ3. C_i denotes one of C_1, C_2, or C_3. 74
4.7 The flow of the experiment for RQ4. 79
4.8 Comparison of the four metrics between repaired and non-repaired samples (robustness, fairness, and safety properties). Each radar plot represents the average of the four metrics for the repaired and non-repaired samples across all datasets. 85
4.9 Comparison of the four metrics between broken and non-broken samples (robustness, fairness, and safety properties). Each radar plot represents the average of the four metrics for the broken and non-broken samples across all datasets. 86

5.1 Our study design. 101
5.2 Empirical cumulative distribution function of EXAM scores for real-world faults (Bugs4Q) and artificial faults (BugsAqua). The filled areas indicate the range between the best-case and worst-case EXAM scores. 106

6.1 Overview of the experiment. 115
6.2 Successful repairs by prompt configuration. 121

List of Tables

2.1 Characteristics of the four repair methods. 11

3.1 Studied SBFL formulae. 25
3.2 Suspiciousness scores in Example 3.4.2. 26
3.3 ANS scores in Example 3.5.1. 26
3.4 Number of data samples for each dataset. 31
3.5 Prediction performance of trained models. 33
3.6 The percentage of settings where the average prediction performance on PAFL-selected data is worse than that on randomly selected data for each dataset. 37
3.7 The percentage of settings where the average prediction performance on PAFL-selected data is worse than that on randomly selected data for each SBFL formula. 37
3.8 The percentage of settings where the average prediction performance on PAFL-selected data is worse than that on SOTA FL-selected data for each dataset. 39
3.9 The percentage of settings where the average prediction performance on PAFL-selected data is worse than that on SOTA FL-selected data for each SBFL formula. 39
3.10 Execution time for PAFL and SOTA FL. 41
3.11 Percentages of data samples in critical ANS ties. Bolded percentages indicate k that leads to the fewest critical ANS ties for each variant and dataset. 44

4.1 Performance and execution time of the three repair methods. 54
4.2 The definition of repairs and breaks through samples. 56
4.3 The definition of repairs and breaks through a property function. 63
4.4 Neural network models for each task. 63
4.5 The number of samples and folds for each dataset. 67
4.6 Effect sizes and U-test results for the differences in metrics between (a) repaired and non-repaired samples and, (b) broken and non-broken samples. 70

4.7 The result of the selection of the most promising repair methods for each case in each dataset and the time saved by our method. 76
4.8 The performance preservation (Trans_{pp}) of repairs and breaks prediction models. A dash ("-") indicates that the Trans_{pp} is not calculated because the source and target are the same. "N/A" indicates that the Trans_{pp} could not be calculated because the f1-score could not be calculated due to bias in the prediction results. 81
4.9 The prediction consistency (Trans_{pc}) of repairs and breaks prediction models. . . 82
4.10 The repair method applied to each property in our experiment. 83
4.11 Prediction performance of repair prediction models for each property and method. For each property and method, we show the average performance value across all datasets. 87
4.12 Prediction performance of break prediction models for each property and method. For each property and method, we show the average performance value across all datasets. 88
4.13 Number of selected promising repair methods and actual optimal repair methods in terms of (a) robustness and (b) fairness. The number in the parenthesis in the sum row represents the total number of cases where each repair method is optimal in terms of correctness. 89
4.14 The performance of the repairs prediction model for different resampling strategies. 93
4.15 The performance of the breaks prediction model for different resampling strategies. 94
4.16 The prediction performance (f1-score) and transferability of the repairs prediction model trained form the combined repair history. R_{src_1} and R_{src_2} are repair methods used to train the model, and R_{tgt} is not used for training. "N/A" indicates that the f1-score could not be calculated due to bias in the prediction results. 95
4.17 The prediction performance (f1-score) and transferability of the breaks prediction model trained form the combined repair history. R_{src_1} and R_{src_2} are repair methods used to train the model, and R_{tgt} is not used for training. "N/A" indicates that the f1-score could not be calculated due to bias in the prediction results. 96

5.1 Numbers of behavior-changing mutants (shown in the column #Mutants (B.-C.)). The column Avg. per Op. is calculated as #Mutants (B.-C.) divided by #Ops. . . . 107
5.2 Comparison of EXAM scores between MBFL and SBFL methods. In the "Sig. level" column, "**" and "-" indicate $p < 0.01$ and $p > 0.05$, respectively. 110

6.1 Repair success rates by prompt configuration and bug type. WO: Wrong Output, TE: Throw Exception. The maximum value in each column is highlighted in bold. 118

6.2 Quality of explanation for each prompt configuration. ↑ (↓) indicates that a higher (lower) value is better. The best value in each row is highlighted in bold. 120

Chapter 1

Introduction

This chapter introduces the motivation and overview of this dissertation. Section 1 discusses the fundamental challenges in debugging AI-enabled and quantum software systems that motivate this research. Section 2 provides an overview of the main contributions of this dissertation, outlining how each addresses specific debugging challenges in AI-enabled and quantum software systems. Section 3 describes the organization of this dissertation.

1 Motivation

Software has become ubiquitous and economically significant in modern society. According to the World Intellectual Property Organization (WIPO) Global Innovation Index 2025 [20], global software spending reached USD 675 billion in 2024, representing a 50% increase from 2020. To further push the boundaries of software capabilities, software systems enhanced with artificial intelligence (AI) and quantum computation are becoming increasingly prevalent. This dissertation refers to them as *AI-enabled software systems* [140] and *quantum software systems* [213], respectively. AI-enabled software systems include safety-critical or economically significant systems (e.g., autonomous driving [28, 202, 47], medical image analysis [106, 64, 170], and chatbots [137, 55, 14]). Misbehavior of these systems can lead to serious accidents or substantial financial losses; for example, the OECD AI Incidents and Hazards Monitor [136] records a case in which Tesla's Autopilot was found partly liable for a fatal crash, leading to a jury verdict of up to USD 329 million in damages [130]. Quantum software systems include applications such as optimization [5] and cryptography [17, 117]. While quantum software systems are not yet as widespread as AI software systems, big tech companies have released open source software (OSS) libraries for quantum software, such as Qiskit [76] by IBM, Cirq [32] by Google, and Q# [120] by Microsoft. These emerging types of software demand software engineering practices as conven-

tional software systems do.

Debugging, the process of isolating and correcting mistakes in computer software [79, 167], is a fundamental software engineering practice for ensuring software quality. It can account for a significant portion of development costs, reaching up to 50–75% in typical commercial organizations [62]. This process includes a variety of interleaved activities such as editing, testing, inspecting, navigating, and consulting external resources [10]. Developers continuously switch among these activities to identify the root causes of faults, fix them, and verify their fixes. To support this process, the software engineering research community has dedicated considerable effort, including automated fault localization [92, 189, 145] and automated program repair [97, 99, 52, 122, 191, 208].

Despite these efforts, conventional debugging techniques may not be well suited to emerging software paradigms such as AI-enabled and quantum software systems. One reason is that both AI-enabled and quantum software systems share inherent debugging challenges rooted in limited observability and complex intermediate states. For AI-enabled software systems, debugging requires attention not only to the source code but also to the datasets and trained AI models, as their behavior depends on data-driven learning processes [160]. It should be noted that this dissertation focuses on AI-enabled software systems that incorporate machine learning models as core components because this type is more prevalent than other types of AI (e.g., rule-based AI). For quantum software systems, debugging should take into account the quantum mechanical properties of quantum software, such as superposition, no-cloning theorem, and entanglement of quantum bits (qubits) [132]. These challenges motivate the development of novel debugging methodologies tailored to AI-enabled and quantum software systems.

2 Dissertation Overview

This dissertation addresses the challenges of debugging AI-enabled and quantum software systems through four complementary studies. Figure 1.1 shows an overview of this dissertation. Unlike conventional software systems, the behavior of AI-enabled and quantum software systems cannot be fully understood from source code alone due to their limited observability and complex internal states that are difficult for developers to inspect directly. Thus, the overarching approach of this dissertation is to adapt conventional debugging techniques to AI-enabled and quantum software systems by incorporating awareness of their component-specific properties. Specifically, all of these studies address either fault localization or repair for AI-enabled or quantum software systems. Among debugging activities, this dissertation focuses on fault localization and repair because they are the most demanding tasks in the debugging process. The following

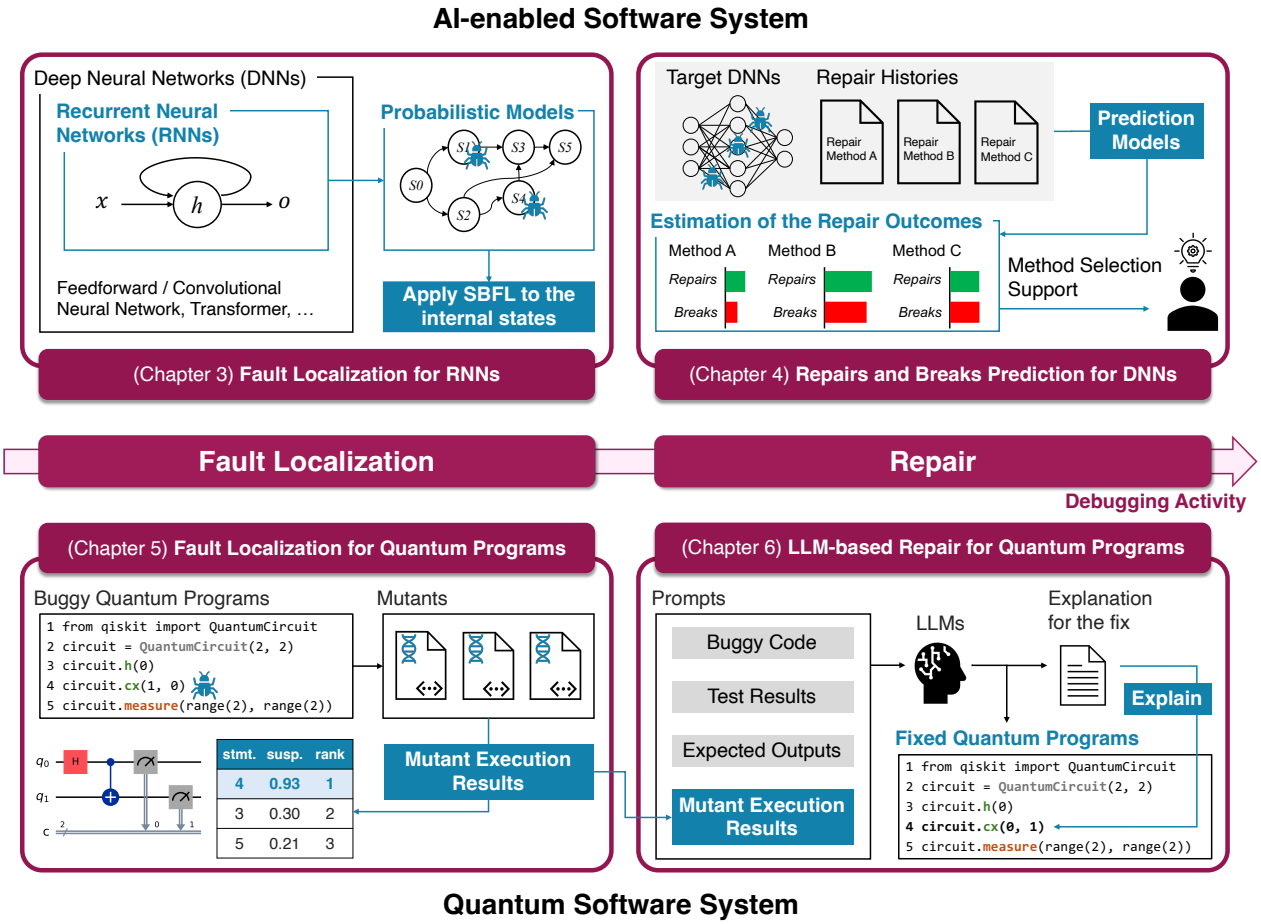


Figure 1.1: Overview of this dissertation.

summarizes the overview of each study, including the challenges, approaches, and key results.

(Chapter 3) PAFL: Probabilistic Automaton-based Fault Localization for Recurrent Neural Networks. This chapter proposes probabilistic automaton-based fault localization (PAFL) to identify faults in recurrent neural networks (RNNs). Among various types of neural networks, RNNs are particularly challenging to debug because they maintain internal states that dynamically change depending on input sequences, unlike feedforward or convolutional neural networks. PAFL adapts the concept of spectrum-based fault localization (SBFL), which analyzes differences in the execution paths of source code, to the sequence of states in RNNs. Specifically, it converts RNNs into probabilistic finite automata (PFAs) and localizes faulty sequences of state transitions on PFAs using the formulas of SBFL. The experimental results demonstrate that PAFL can identify data samples leading to misbehavior more effectively and efficiently than the state-of-the-art fault localization technique (i.e., RNNRepair [194]), achieving better performance in 66.74% of the

cases. This suggests that PAFL is useful for debugging AI-enabled software systems that include RNNs.

(Chapter 4) Repairs and Breaks Prediction for Deep Neural Networks. This chapter tackles the challenge of predicting the outcomes of repair methods for deep neural networks (DNNs). Here, repair methods refer to those that target components of the DNN model itself (e.g., neurons or weights). These methods aim to reduce the specific type of misbehavior of the DNN while preserving its existing correct behavior [111, 45, 205, 200, 171, 165, 177, 149, 23, 104, 72, 174]. Preliminary experiments in this study revealed that the most effective repair method varies depending on the dataset, model architecture, and repair perspective. Therefore, by predicting the outcomes in advance, developers can select more effective repair methods and reduce the number of trial-and-error iterations during the debugging process of AI-enabled software systems. Specifically, we build two types of prediction models: (1) predicting whether a misclassification can be corrected by a repair method, and (2) predicting whether a correct classification can be changed to misclassification by a repair method. These prediction models use the confidence and ambiguity of the pre-repaired DNN outputs for each sample as explanatory variables. The experimental results demonstrate that in the repair application scenario, the prediction models yield an average time savings of 16.29% compared to the naive method of applying all repair methods and checking the results. This suggests that the prediction models are useful for developers and operators of DNNs as they assist in deciding whether or not to apply a repair method and, if so, which method to use.

(Chapter 5) Evaluating Mutation-based Fault Localization for Quantum Programs. This chapter investigates the effectiveness of mutation-based fault localization (MBFL) for quantum programs. Quantum programs, which form the core of quantum software systems, consist of operations on qubits, where each operation is referred to as a quantum gate. The effectiveness of mutation testing for quantum programs that apply mutations to quantum gates (e.g., deleting quantum gates) has been demonstrated [11, 118, 50, 179, 119]. We believe that these operations hold promise for enhancing fault localization in quantum programs. While MBFL was originally proposed for classical (non-quantum) programs, we first evaluate its effectiveness in the quantum context. The evaluation involves 23 real-world faults and 305 artificial faults in quantum programs developed with Qiskit, a representative Python library for writing quantum programs. The experimental results demonstrate that (1) real-world faults are more challenging for MBFL than artificial faults, (2) MBFL is more effective than SBFL, and (3) mutation operations specific to quantum programs are more effective than classical mutation operations. This study presents

the empirical evidence that MBFL is promising for debugging quantum programs.

(Chapter 6) Extended Work: Leveraging Mutation Analysis for LLM-based Repair of Quantum Programs. This chapter explores the use of mutation analysis results to assist large language models (LLMs) in repairing faulty quantum programs. This study assumes MBFL (introduced in Chapter 5) as a preceding step in the repair process for quantum programs. An existing study has attempted to utilize LLMs for repairing quantum programs [59]. Meanwhile, in the context of classical programs, it has been shown that the information included in prompts can significantly affect the performance of automated program repair [192, 21, 199, 43]. In this study, we construct a framework in which a LLM generates code repairs along with a natural language explanation of the applied repairs. We hypothesize that mutation analysis results can improve the repair success rate and help the model produce richer explanations, as they provide more detailed dynamic information than simple execution outputs, such as stack traces. Our evaluation demonstrates that incorporating mutation analysis results into the prompt enabled the LLM to repair 17 out of 18 faulty programs. Moreover, mutation analysis results improve explanation quality by making positional descriptions more accurate, complete, and concise. This study demonstrates the effectiveness of combining MBFL and LLMs for debugging quantum programs.

3 Dissertation Organization

The rest of this dissertation is organized as follows:

- Chapter 2 reviews the related work.
- Chapter 3 introduces PAFL, a fault localization method for RNNs.
- Chapter 4 introduces repairs and breaks prediction for DNNs.
- Chapter 5 investigates the effectiveness of MBFL for quantum programs.
- Chapter 6 explores the use of MBFL results to assist LLMs for repairing faults in quantum programs.
- Chapter 7 concludes this dissertation.

Chapter 2

Related Work

This chapter introduces research related to this dissertation. Specifically, it reviews debugging techniques for conventional software, AI-enabled software, and quantum software in separate sections. Each section is organized into subsections focusing on *fault localization* and *repair*. In this dissertation, fault localization refers to the process of identifying the specific locations of faults in software components, while repair refers to the process of modifying these components to fix the identified bugs.

1 Debugging Conventional Software

In this dissertation, *conventional software* refers to software systems that do not incorporate specific components such as AI models or quantum programs. Since the debugging techniques for conventional software serve as a fundamental basis for developing and understanding debugging approaches for AI-enabled and quantum software, this section provides a comprehensive review of these techniques.

1.1 Fault Localization for Conventional Software

Traditional fault localization techniques include program logging (e.g. `print` and `logging` statements), assertions, and breakpoints [204]. Assertions are used to verify that specific conditions hold true at runtime [154]. They are provided as built-in features in programming languages such as Java and Python. On the other hand, breakpoints allow developers to suspend program execution to inspect the internal state. These are typically integrated into integrated development environments (IDEs) such as Visual Studio Code.[1]

[1] https://code.visualstudio.com/

Program	t1	t2	t3
1 def check_val(val):	✓	✓	✓
2 if not type(val) is int:	✓	✓	✓
3 return "even"	✓		
4 elif (val%2)==0:		✓	✓
5 return "even"		✓	
6 else:			✓
7 return "odd"			✓
Test results	Fail	Suc.	Suc.

Figure 2.1: Example faulty program and execution traces for three test cases.

As software systems grow in scale, the application of these traditional fault localization techniques becomes increasingly inefficient. Consequently, automated fault localization techniques have been proposed [189]. These techniques include slicing-based [39, 116, 95], spectrum-based [7, 8, 6, 9, 145, 186], mutation-based [145, 123, 143], machine learning-based [190, 188], and information retrieval-based [218, 155, 209] approaches.

- *Spectrum-based fault localization (SBFL)* [7, 8, 6, 9, 145, 186] identifies faulty statements based on program execution traces collected during testing. Figure 2.1 illustrates a simple faulty program and its execution traces for three test cases. By using this information (i.e., how many times each statement is executed in passing and failing tests), SBFL calculates the suspiciousness score for each statement.

- *Mutation-based fault localization (MBFL)* [145, 123, 143] extends SBFL by incorporating test results derived from mutation analysis in addition to execution traces. Specifically, MBFL considers the impact of mutation operations (i.e., syntactic modifications such as altering logical operators) on each statement and whether applying a mutation causes the test outcome to change from pass to fail, or vice versa.

Pearson et al. [145] compared the performance of SBFL (using five different formulas, e.g., Ochiai [8]) and MBFL (MUSE [123] and Metallaxis [143]) on 2,995 artificially injected faults and 310 real-world faults in Java programs. They reported that while MBFL exhibited high performance for artificial faults (injected through mutation operations), it was less effective than SBFL in detecting real-world faults. These findings suggest that SBFL and MBFL possess distinct strengths depending on the nature of the target faults. In our study, we adopt SBFL and MBFL as fault localization

techniques for AI-enabled software and quantum software. SBFL is chosen because it is one of the most widely studied fault localization techniques for conventional software [189]. MBFL is chosen because it is reported to have strengths different from SBFL in the evaluation by Pearson et al. [145]. We investigate whether these fault localization techniques remain effective when applied to AI-enabled software (Chapter 3) and quantum software (Chapter 5).

Driven by the rapid evolution of LLMs, a new generation of fault localization techniques has emerged [84, 198, 150, 195]. Yang et al. [198] introduced *LLMAO*, a language model-based fault localization approach that requires no test execution information. They trained a language model that outputs suspiciousness scores for each line by leveraging the internal representations of the CodeGen models [133]. *AUTOFL* [84] utilizes function calling to enable the LLM to conduct project-level fault localization (i.e., fault localization across multiple files). Notably, it not only pinpoints the fault location but also generates an explanation of the bug. Similarly, *SOAPFL* [150] targets project-level fault localization. It constructs an LLM-driven standard operating procedure (SOP) to automatically localize buggy methods across the entire software. This process is structured into three distinct phases: fault comprehension, codebase navigation, and fault confirmation, each of which is executed by LLMs.

1.2 Repair for Conventional Software

Given that debugging accounts for a dominant portion of software development costs [62], *automated program repair (APR)* [52] is a promising approach to alleviate this burden. APR techniques can be broadly classified into three categories [100]:

1. *Heuristic-based repair:* This approach iteratively generates and validates patches. For instance, GenProg [98] employs genetic programming to generate patches. Specifically, it maintains a population of variants of the target faulty program generated by mutation operators. It then selects variants to maximize a fitness function calculated from the number of passing test cases. At a high level, GenProg iterates this cycle to evolve program variants that pass as many tests as possible.

2. *Constraint-based repair:* This approach formulates repair as a constraint-solving problem. A representative example is SemFix [131], which is built upon symbolic execution and program synthesis. It derives repair constraints from the test suite and utilizes an SMT solver to synthesize a program that satisfies these constraints.

3. *Learning-based repair:* This approach builds predictive models to generate patches. For instance, DeepFix [60] is an early approach targeting C programs. It utilizes a multi-layered

sequence-to-sequence neural network with attention mechanism.

Among these approaches, learning-based repair has recently gained attention due to the rapid progress of machine learning techniques, particularly LLMs [208]. Xia et al. [191] directly applied LLMs to APR and demonstrated that LLM-based APR can outperform conventional APR techniques for Java, Python, and C. Since then, numerous studies have explored LLM-based APR [78, 192, 21, 43]. For example, InferFix [78] enhances prompts with static analysis results and bug locations, enabling the LLM to generate repairs with explicit awareness of the bug type and its position in the code. Ehsani et al. [43] enhance prompts by hierarchically injecting bug-, repository-, and project-level knowledge to provide broader and more relevant repair context. RepairAgent [21] is an autonomous agentic APR approach based on LLMs. It employs *dynamic prompting*, allowing the agent system to decide which information to include in the LLM prompt and dynamically update it through function calling. Their evaluation demonstrated that RepairAgent autonomously repaired 164 bugs in Defects4J [83], including 39 bugs that were not fixed by prior techniques.

2 Debugging AI-enabled Software

AI-enabled software relies on data-driven logic, which introduces unique challenges such as its black-box nature and the test oracle problem [160, 12, 206]. Empirical studies have examined the characteristics of AI-specific bugs [75, 71]. These studies indicate that debugging AI-enabled software requires addressing faults in data and models, not just code. Among machine learning models, DNNs are particularly common, prompting substantial research on debugging techniques tailored to software that incorporates DNNs. This section reviews these approaches and how they address the unique challenges of AI-enabled software.

2.1 Fault Localization for AI-enabled Software

Eniser et al. [45] proposed DeepFault. They sought to identify the root causes of misbehavior not in the source code but within the DNN model itself. Specifically, DeepFault applies SBFL to DNN models to identify the *suspicious neurons*, which contribute to misbehaviors of DNN models. It records neurons that are activated when a test sample is successfully or unsuccessfully predicted. Here, a neuron is considered activated when the value of the activation function of each neuron exceeds a predefined threshold. Based on these records, suspiciousness scores are calculated for each neuron. In DeepFault, neurons with high suspiciousness scores are identified as suspicious neurons. Wardat et al. [183] proposed DeepLocalize, a dynamic fault localization approach that inspects the execution traces of DNN training, including weights, gradients, and

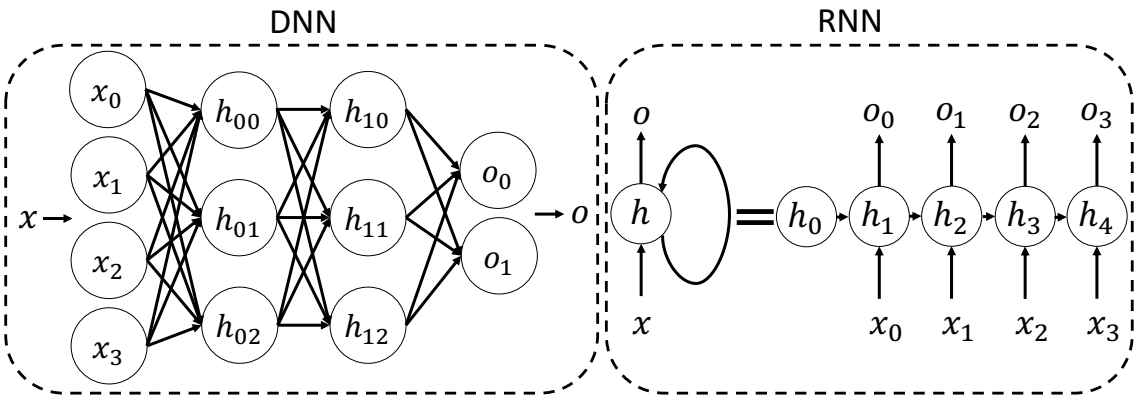


Figure 2.2: Differences between DNN and RNN.

loss values. By analyzing the historical trends of these values, it detects numerical anomalies such as vanishing gradients or saturated activation functions. Consequently, DeepLocalize identifies the specific faulty layers exhibiting these anomalies or the hyperparameters responsible for the training failure. Building on this, Wardat et al. [182] developed DeepDiagnosis, which aims to explain the root causes of model failures rather than merely locating faulty components. It monitors the training process to detect high-level symptoms (e.g., loss not decreasing) and maps them to specific faults (e.g., an excessively high learning rate) using decision-tree-based logic. Ghanbari et al. [53] proposed Deepmufl, which applies MBFL to DNN models. Deepmufl identifies the faulty layers of a model by utilizing model-level mutation operators [110, 67], such as removing dense layers or changing activation functions. Their evaluation on 109 bugs collected from Stack Overflow demonstrated that Deepmufl successfully localized 53 bugs (48.6%) at the top-1 position, outperforming existing fault localization techniques for DNNs, such as DeepLocalize and DeepDiagnosis.

Despite these extensive studies, few approaches are capable of handling RNNs, a specific type of DNN. RNNs are well suited for processing sequential data such as text and audio data. As shown in Figure 2.2, they update their internal states dynamically in response to input sequences. This dynamic behavior makes fault localization for RNNs more challenging than for other DNN architectures. To capture this characteristic of RNNs in a simplified form, existing studies have proposed extracting alternative abstract models from RNNs [184, 42, 40, 66, 41, 180]. However, these approaches have been aimed at behavior explanation or robustness analysis, and have not been applied to fault localization. Therefore, in Chapter 3, we set out to develop a fault localization technique for RNNs that addresses this challenge.

Table 2.1: Characteristics of the four repair methods.

Method	Target	Perform localization?	Technique for repairing	Evaluated architecture
CARE [171]	Neurons	Yes	Particle swarm optimization	FNN and CNN
Apricot [205]	Weights	No	Retraining	CNN
Arachne [165]	Weights	Yes	Differential evolution	CNN and RNN
APRNN [174]	Weights	No	Linear programming	FNN and CNN

2.2 Repair for AI-enabled Software

Similar to fault localization, a significant body of research on automated repair has focused on DNN models. Existing repair methods for DNN models can be broadly classified into two approaches: (1) repairing the DNN models themselves (e.g., modifying weights or neurons) [111, 45, 205, 200, 171, 165, 177, 149, 23, 104, 72, 174], and (2) repairing the source code used to build the models [183, 212, 182, 24, 89, 53]. Since this dissertation focuses on model-level repair, we primarily describe related studies in that direction.

Table 2.1 shows the characteristics of four representative model-level repair methods: CARE [171], Apricot [205], Arachne [165], and APRNN [174]. The table summarizes them in terms of the target component for repairing, whether the method includes localization, the technique used for repairing, and the architecture evaluated in the original paper.

CARE [171] is a repair method that directly modifies the neuron values of neural networks to satisfy desired properties with minimal changes. It uses causal analysis to localize the neurons that cause misbehavior and reduce the search space for repair. These neuron values are then modified by Particle Swarm Optimization (PSO) [87], a continuous optimization method. CARE's effectiveness was evaluated for FNNs on tabular datasets and CNNs on image datasets, demonstrating improved accuracy, fairness, and safety. Apricot [205] is a repair method that leverages *reduced models* trained on a subset of the original training set. The weights of the reduced models are used to adjust the weights of the original model. First, the reduced models are divided into two groups: those that correctly classify the input sample and those that misclassify it. Then, the weights of the original model are adjusted to be close to the former group and far from the latter. Experimental results using three different CNNs on image datasets showed increased accuracy with Apricot. Arachne [165] is a repair method that directly modifies the neural network weights, but the method is different from CARE. They propose *bidirectional fault localization* to identify suspicious weights with a high impact on output and responsibility for misbehavior. To modify these weights, Arachne uses Differential Evolution (DE) [169], a population-based optimization algorithm. Arachne successfully repaired common types of misbehavior in CNNs on

three image datasets and demonstrated improved fairness in a Long Short-Term Memory (LSTM) model, which is a type of RNN. APRNN [174] is a provable repair method in the sense that it ensures that repaired DNNs satisfy the given repair specification. Here, the repair specification is the output condition that the input sample of a particular condition should satisfy. In a nutshell, APRNN transforms the repair specification into constraints for linear programming (LP) and finds a minimal weight change that satisfies the constraints by solving the LP (using a solver such as Gurobi [61]). APRNN's effectiveness was evaluated for FNNs on safety datasets and CNNs on image datasets. They evaluated not only the generalization, but also the drawdown (corresponding to breaks in our study) of APRNN, and some degree of drawdown was reported for unseen data samples.

While numerous repair methods have been proposed, only a few studies have compared these methods and conducted in-depth investigations to apply them in the development and operation of software systems using DNNs. Song et al. presented AIRepair [166], a platform that integrates multiple repair methods into one easy-to-use tool. This enables both researchers and developers to try out repair methods. They also compared the performance of existing repair methods using the tool. Kim et al. conducted an empirical study of three repair methods for *model architecture faults* in DNNs [89]. These faults refer to errors in specifying the model structure and training process in the source code. The evaluated repair methods include AutoTrainer [212] for source code, as well as hyperparameter optimization methods proposed in the ML community [36, 46]. They reported that these methods sometimes show comparable performance with a random baseline.

Despite these extensive efforts on repair methods for DNN models, existing studies have not sufficiently addressed how developers should practically choose among them. Indeed, our preliminary investigation (see Section 2 of Chapter 4) revealed that the most suitable repair method varies depending on the model, dataset, and evaluation perspective (i.e., accuracy, robustness, fairness, and safety). To address this gap, Chapter 4 analyzes the characteristics of the repairs and breaks introduced by each repair method and builds a model that predicts these outcomes in advance. This predictive model enables developers to estimate both the expected benefits and potential side effects before applying a repair, allowing them to select the most appropriate method for their specific scenario.

3 Debugging Quantum Software

Since *quantum software engineering (QSE)* emerged as a distinct research area in software engineering around 2020 [213], numerous studies have been conducted in this field. The significance of this area is underscored by its recognition as one of the seven landmarks in the *Software Engi-*

```
1 from qiskit import QuantumCircuit
2 circuit = QuantumCircuit(2, 2)
3 circuit.h(0)
4 circuit.cx(0, 1)
5 circuit.measure(range(2), range(2))
```

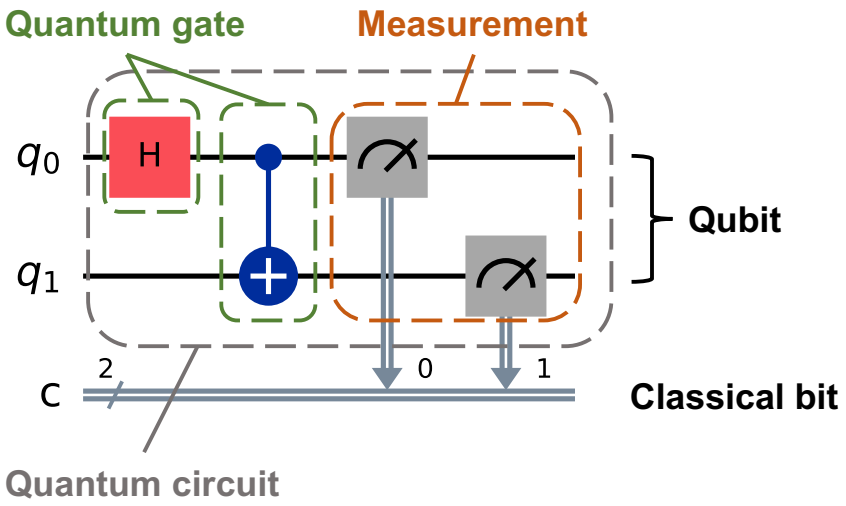


Figure 2.3: An example of a quantum program written in Qiskit (above) and corresponding quantum circuit (below).

neering 2030 roadmap [125, 102, 148]. In particular, debugging quantum programs has established itself as a critical topic within this landscape, driving a significant body of research efforts [102]. Quantum programs form the core of quantum software and describe quantum circuits such as the one shown in Figure 2.3. A quantum circuit consists of defining qubits, applying quantum gates, and measuring qubits. Numerous studies report that these quantum-specific elements are a major source of quantum program bugs [70, 216, 141, 107, 215]. For example, Luo et al. [107] collected and analyzed 96 real-world quantum program bugs and their fixes. They found that more than 80% of the bugs were quantum-specific in sense that they required quantum-domain knowledge to resolve. Their bug categories include incorrect qubit initialization, wrong qubit ordering, incorrect gate application, and incorrect measurement.

3.1 Fault Localization for Quantum Software

Sato and Katsube [158, 159] identified four characteristics specific to quantum program testing (e.g., the cost of testing a specific part of a quantum program depends on its location) and proposed a fault localization method that accounts for these characteristics. Their approach con-

structs a cost-based binary search tree from a quantum program and narrows down the testing scope using this tree to identify faulty locations. This tree is built by treating the quantum program as a sequence of quantum gates. However, real-world quantum programs, such as those in Bugs4Q [214], often incorporate classical instructions (e.g., variable declarations and control structures) in addition to quantum instructions (e.g., quantum gate declarations and measurements). As a result, representing a quantum program purely as a sequence of quantum gates may not always be feasible. In contrast, mutation analysis can be applied to quantum programs containing both classical and quantum instructions [50]. Thus, we consider MBFL to have broader applicability in Chapter 5.

3.2 Repair for Quantum Software

Repairing quantum programs is inherently challenging because it requires deep domain-specific knowledge of quantum computing [162, 107, 203]. As a pioneering work on LLM-based APR for quantum programs, Guo et al. [59] utilized ChatGPT [137] to explore APR for quantum programs. Their results showed that while ChatGPT successfully repaired 97% of classical (non-quantum) bugs in their dataset, it only managed to repair 17% of quantum bugs. Another line of research focuses on *synthesis-based APR* techniques, which generate and insert new gates to fix buggy programs. For instance, Li et al. [103] proposed UnitAR, which repairs quantum programs by automatically generating unitary operations. Similarly, Tan et al. [173] proposed HornBro, which removes faulty gates responsible for the bug and synthesizes new ones to produce the correct behavior. They demonstrated that HornBro could fix more bugs than the ChatGPT-based approach. As a side effect, however, HornBro introduces many additional gates, generating up to 249 new gates. LLM-based and synthesis-based APR approaches exhibit different characteristics. In Chapter 6, we aim to improve LLM-based repair with mutation analysis. We chose an LLM-based repair approach for the following reasons:

1. Unlike synthesis-based approaches, which are limited to gate-related bugs, LLM-based approaches are more flexible and can handle a wider range of bugs such as API-related bugs, which are also commonly found in real-world quantum programs [107].

2. Synthesis-based approaches increase the number of gates in a quantum program, thereby increasing its complexity. Avoiding excessive gate counts is recommended as a best practice by Google Quantum AI [56] and has also been identified as a type of quantum code smell [29].

Our approach not only instructs the LLM to generate minimal repairs but also to provide natural

language explanations for them. Unlike other APR techniques for quantum programs, this allows developers to examine and validate the rationale and intent behind each change.

Chapter 3

PAFL: Probabilistic Automaton-based Fault Localization for Recurrent Neural Networks

1 Introduction

Neural network models have been incorporated into safety-critical systems, such as self-driving cars [28, 19, 47, 58], malware detection [201, 176, 90, 85], and medical image analysis [106, 163, 64] in recent years. For example, a previous study used convolutional neural networks (CNNs) to automatically control the steering wheel of self-driving cars from front camera images [19]. If such safety-critical systems are unreliable, *misbehavior* may occur, leading to serious accidents, such as cars crashing into a tree owing to incorrect control of steering wheels [4, 3]. Hence, it is expedient to improve the reliability of neural network models. In this study, we refer to the causes of such misbehavior *faults*.

Various methods have been developed to localize and correct faults in neural network models to improve reliability [206, 147, 111, 194, 45, 165, 69, 88, 109, 175, 135, 172, 108, 110, 57, 183, 205, 200, 42, 68]. For example, Arachne [165] repaired neural network models by adjusting the weights that cause misbehavior using a search-based approach. DeepFault [45] modified faulty neurons that cause misbehavior by retraining the models rather than adjusting the weights.

The identification and correction of faults in recurrent neural networks (RNNs) are essential; however, they are more difficult to achieve than for other neural network models. One reason for this is the difficulty of proposing *fault localization (FL) techniques* for RNNs. Other neural network models generally have static (unchanged) components, such as weights, and FL identifies faulty components. Conversely, RNNs dynamically change their internal states depending on the input data samples (e.g., words on sentences), and FL must cover all and sometimes enormous snapshots

of such states to localize states corresponding to faults.

Researchers can use approaches that transform RNNs into static models to address this challenge. For example, previous studies have proposed approaches that transform RNNs into *deterministic finite automata (DFAs)* [184, 66] and *probabilistic finite automata (PFAs)* [40]. These approaches relieve this challenge and enable FL techniques to be applied to RNNs. In this paper, we propose an FL technique using this approach to improve the reliability of RNNs.

We propose a novel FL technique called *probabilistic automaton-based fault localization (PAFL)* for RNNs. The main steps of PAFL are as follows: transforming RNNs into PFAs [40], and localizing the *sequences of state transitions* on PFAs as faults of RNNs with n-grams. The key idea is to localize faults in the sequences of state transitions on PFAs with n-grams. A state transition indicates a transition from one state to another on the PFAs, which represents a part of the snapshots of the internal states of RNNs. Hence, sequences of state transitions, which are n-grams of state transitions on PFAs, can represent more snapshots and would localize faults more precisely. We use the term n-grams, which is inspired by natural language processing. This is because n-grams originally means considering the sequences of words, and we suppose considering sequences of state transitions on PFAs is similar to it. Additionally, we distinguish data samples that contribute to the misbehavior of RNNs. We propose a novel suspiciousness score, the *average n-gram suspiciousness (ANS) score*, based on n-grams and PFAs (sequences of state transitions) to distinguish data samples that contribute to misbehavior. This score could indicate the degree to which each data sample contributes to the misbehavior of RNNs if PAFL accurately localizes faults on sequences. Hence, this step is used to evaluate whether PAFL accurately localizes the sequences indirectly.

We evaluate the effectiveness of PAFL and ANS scores on eight public datasets. Four of the eight are artificial language datasets, and the others are real-world datasets comprising movie-review texts, comments on Wikipedia talk pages, and handwritten digit images. Furthermore, we use three RNN variants as a representation of RNNs: simple recurrent neural networks (SRNNs), gated recurrent units (GRUs), and long short-term memories (LSTMs). We investigate the following research questions (RQs). RQ1 examines whether the proposed ANS score can measure the degree to which each data sample contributes to the misbehavior of RNNs. RQ2 examines whether data samples identified by PAFL contribute to the misbehavior of RNNs.

RQ1) Do ANS scores represent the misbehavior of RNNs?

ANS scores represent the misbehavior of RNNs. ANS scores tend to overlook misbehavior when n of n-grams is one; however, they capture it when n is greater than one. Hence, adopting the concept of n-grams by considering a sequence of multiple states is important.

RQ2) Does PAFL identify data samples that cause the misbehavior of RNNs?

The prediction performance of RNNs on data samples selected by PAFL is statistically significantly lower with large effect sizes from randomly selected data samples in 43.44% of the settings (15,638/36,000) and state-of-the-art fault localization in 66.74% of the settings (961/1,440). Hence, PAFL distinguishes data samples that cause the misbehavior of RNNs. Specifically, PAFL works well in the datasets where RNNs exhibit poor performance.

The contributions of this study are as follows:

- We propose PAFL, a fault localization method for RNNs. PAFL extracts data samples that are difficult to predict, leading to misbehavior, in RNNs. The results reveal that PAFL works well when developers have to inspect RNNs (i.e., when RNNs exhibit poor performance).
- We propose an ANS score to quantify the degree of contribution to misbehavior for each data sample. Furthermore, we demonstrate the importance of considering the sequences of states of PFAs extracted from RNNs to quantify them.
- We provide a replication package for the experiments on a publicly available repository.[1]

The remainder of this paper is organized as follows. Section 2 describes the challenge of fault localization for RNNs and a solution. Section 3 explains the proposed approach. Section 4 presents the experimental setup. Section 5 presents the experimental results. Section 6 discusses the results of this study. Section 7 explains the threats to the validity. Section 8 summarizes this study.

2 Challenges of Fault Localization for RNNs

In this paper, we propose an approach for conducting fault localization on RNNs using the basic idea of SBFL because it has already been successfully applied to localize faults in neural network models. However, a challenge hinders fault localization on RNNs. In this section, we describe the challenge and how to address it using the proposed approach.

Challenge: RNNs dynamically change their internal states. As described in Section 1, RNNs dynamically change their internal states based on the input data samples. Figure 2.2 shows a DNN model (left) and an RNN model (right). Assume that we feed an input data sample x, which consists of four elements $x = \{x_0, x_1, x_2, x_3\}$, into the model and obtain output o. DNNs have some hidden states (e.g., h_{00}) with fixed parameter values. Conversely, RNNs have an internal state h, whose parameter values are updated based on the sequences of the elements of the input data samples. For example, Figure 2.2 shows that the initial state h_0 is updated by x_0 to h_1.

[1] https://github.com/posl/PAFL-replication

This characteristic makes it difficult for researchers to propose methods for localizing faults in RNNs compared with other architectures. For example, FL techniques for DNNs feed input data samples that result in misbehavior into DNNs to identify neurons through which such input data samples pass, and compute their suspiciousness scores to identify suspicious neurons h_{ij} [45, 111]. The challenge is that even if an FL technique identifies the h_i of RNNs like the h_{ij} of DNNs, the h_i of RNNs changes depending on the input data unlike the h_{ij} of DNNs.

Solution: We propose a novel fault localization approach called *probabilistic automaton-based fault localization (PAFL)* to address this challenge. Specifically, PAFL converts RNNs into PFAs, which are static models. We select PFAs instead of DFAs because a previous study [40] argued that PFAs are superior to DFAs in the model extracted from RNNs. PAFL assumes that such state transitions represent the internal states of RNNs because state transitions in PFAs are the basis for prediction. However, such a state transition represents a part of the snapshots of the internal states of RNNs. Hence, PAFL considers state transitions and sequences of state transitions (n-grams of state transitions) on PFAs and computes their suspiciousness scores to localize faults. Such sequences are better representations of the snapshots of internal states than state transitions, and PAFL would localize faults of RNNs more precisely.

3 Our Approach

This section presents a detailed explanation of PAFL, the proposed fault localization method for RNNs.

3.1 Overview

Figure 3.1 shows the overview of PAFL. In **Phase 1**, we extract a PFA from the trained RNN by applying the Dong et al. [40] method. In **Phase 2**, the PFA is used to collect two types of logs, *successful logs* and *failed logs*. A log includes the sequence of state transitions in PFA for each data sample. The successful log consists of a list of data samples successfully predicted by the PFA and the state transitions on PFA, whereas the failed log consists of a list of data samples unsuccessfully predicted by the PFA and the state transitions on PFA. In **Phase 3**, based on these logs, we compute the suspiciousness score on n consecutive state transitions (i.e., n-grams) using the SBFL on the PFA. In **Phase 4**, we select data samples that may cause the RNN to misbehave to evaluate PAFL. For this selection, we define a new score for each data sample called an *average n-gram suspiciousness score (ANS score)*. The ANS score is calculated based on the suspiciousness score for n-grams computed using the SBFL.

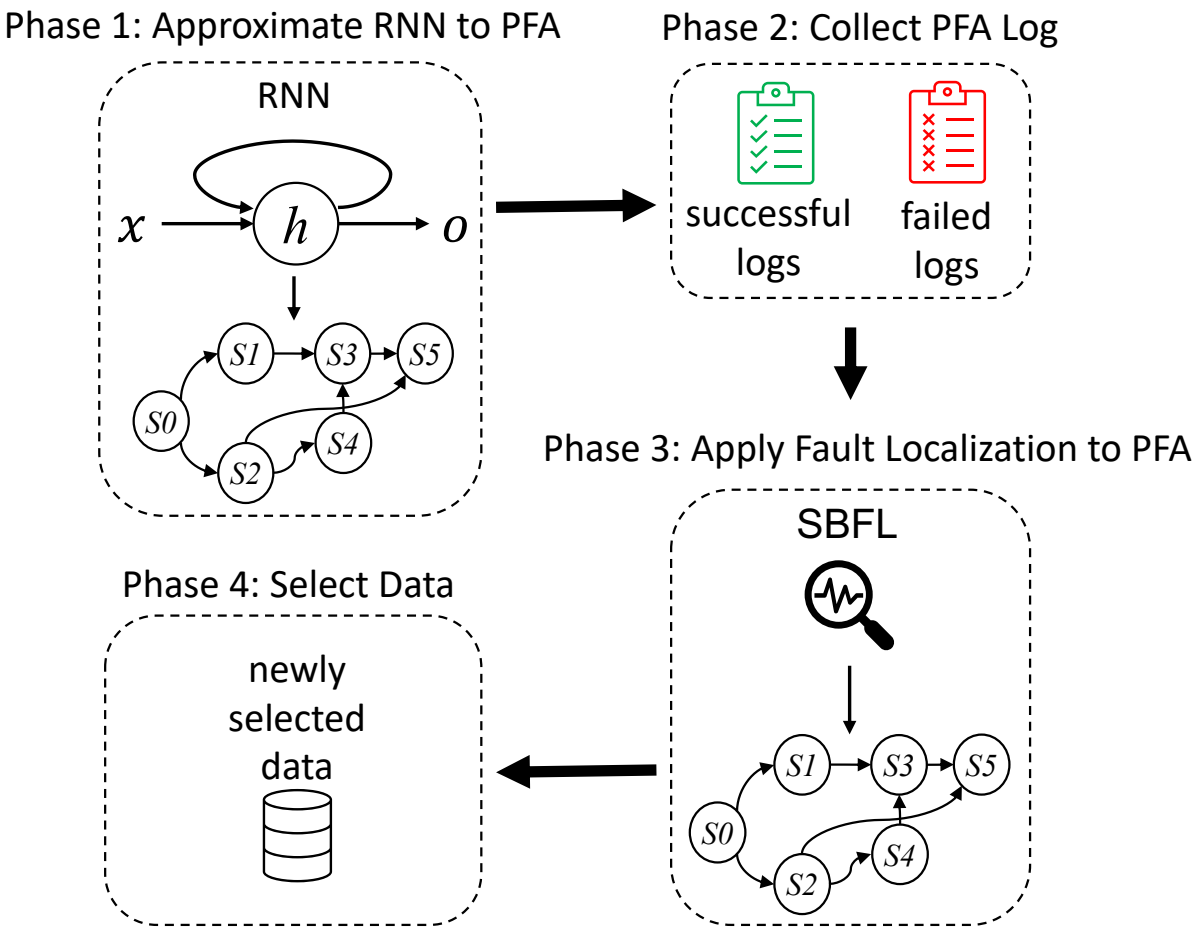


Figure 3.1: Overview of PAFL.

3.2 Phase 1: Approximate RNN to PFA

Recently, researchers have attempted to extract simpler models from complex neural network models, including RNNs, to alleviate their black-box nature. For example, previous studies extracted deterministic finite automata (DFAs) or probabilistic finite automata (PFAs) from RNNs [40, 66, 210, 184]. Dong et al. [40] argued that PFAs are superior to DFAs as models extracted from RNNs. This is because PFAs can make probabilistic decisions whereas DFAs cannot. Hence, we used PFAs in this study.

PFAs [151] are finite automata comprising a finite set of states and probabilistic state transitions. Figure 3.2 shows an example of a structure of a PFA. This figure shows a PFA consisting of six states $\{S_0, \cdots, S_5\}$ and seven probabilistic state transitions $\{P_0, \cdots, P_6\}$. Among the states of the PFA in Figure 3.2, S_0, indicated by the thick arrow, is the initial state. The set of states $F = \{S_4, S_5\}$, surrounded by double circles, indicates the final states. Each state transition P_i is the probability of taking a value between zero and one, indicating the transition probability from

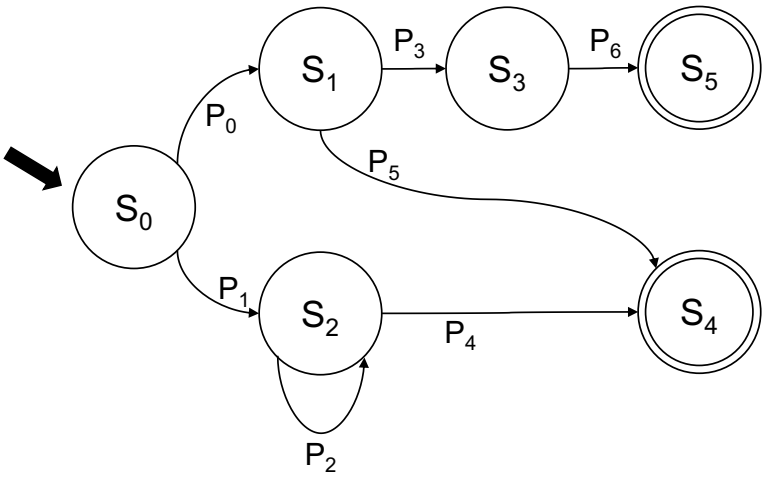


Figure 3.2: An example of a structure of a PFA.

the state at the root of the arrow to the state ahead.

In this study, we extract a PFA from a trained RNN using the method proposed by Dong et al. [40]. This method comprises two parts: *abstraction* and *learning*. In the abstraction part, the hidden states of RNNs are abstracted as *abstract states*. The abstract state is a cluster of hidden states of RNNs obtained by clustering. In the learning part, the *AAlergia* [115] algorithm is used to construct PFA states and state transitions from a set of sequences of abstract states obtained during the abstraction part. The AAlergia algorithm constructs a PFA that abstracts the original RNN based on the abstract states.

Abstraction part. The abstraction part is described in detail. In this part, the hidden states of the target RNN are abstracted by utilizing k-means [113] clustering. In k-means clustering, the value of k indicates the degree of abstraction, with a smaller and larger k indicating coarser and finer abstraction, respectively.

Figure 3.3 shows the flow used to obtain the abstract states from the hidden states ($k = 3$). We describe the flow using data sample x. Notably, the RNN was already trained with X_{train}. First, we collect the sequence of the hidden states of the RNN when feeding x into RNN, $\langle h_0, h_1, h_2, h_3, \cdots \rangle$, as shown in the upper right. We repeat this process for all data samples in X_{train}. Second, we classify all hidden states from all data samples into three clusters using k-means, as shown in the lower right. Each of the k clusters is referred to as an abstract state. Third, for a data sample x, we convert the sequence of hidden states of the data samples x into a sequence of abstract states. The corresponding abstract state of each hidden state is extracted from the clustering result. Consequently, we obtain the sequence of abstract states $\langle 0, 0, 1, 2, \cdots \rangle$ for data sample x.

Learning part. Subsequently, the learning part is described. In this part, the AAlergia algorithm

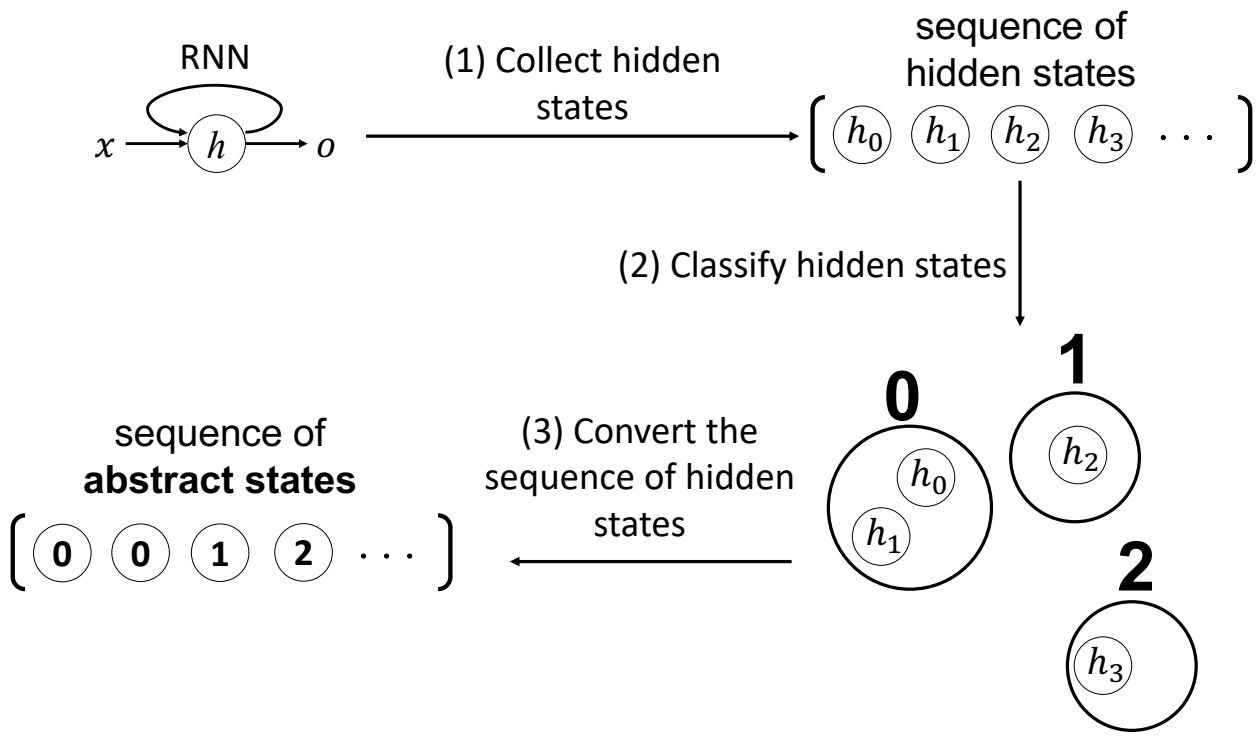


Figure 3.3: Flow of hidden states abstraction ($k = 3$).

is used to construct PFA. The algorithm uses a set of sequences of abstract states as the input. This algorithm uses *prefixes* for each sequence of the input set to construct a tree called *frequency prefix tree (FPT)*. The prefix indicates the first i abstract states for each sequence.[2] The nodes in the tree are iteratively merged to reduce the number of nodes. Once the algorithm has merged sufficient nodes, it constructs a PFA based on the tree. Hence, the number of states and state transitions in the resulting PFA depends on the number of clusters k in the abstraction, and the FPT nodes to be merged in the algorithm. For more details on the algorithm, refer to [115].

3.3 Phase 2: Collect PFA Log

We can collect successful and failed logs on the PFA based on the prediction results of the PFA. The procedure of prediction by the PFA is as follows:

1. We obtain the sequence of hidden states when a sample is fed into the original RNN.
2. We apply the clustering function obtained in the abstraction to the sequence to transform them into a sequence of abstract states.

[2]Let $i = 3$ in Figure 3.3, the prefix would be $\langle 0, 0, 1 \rangle$.

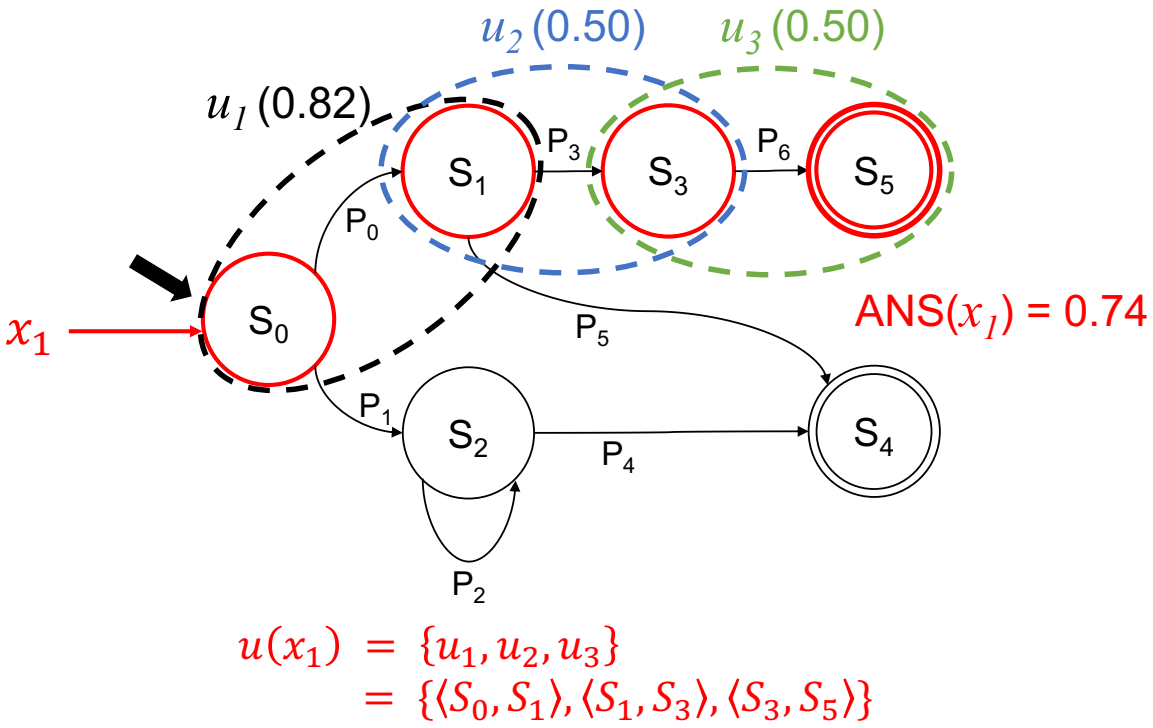


Figure 3.4: The bigrams (i.e., n-grams of $n = 2$) of PFA states obtained by feeding a sample x_1. The parentheses after u_i represent the suspiciousness scores.

3. We follow the transition of the PFA with a sequence of abstract states and obtain the final state s.

4. We label the sample as a label corresponding to the final state s.

We store the transition of the PFA and the input sample in successful or failed logs based on whether the label of the final state s is correct.

Notably, we can measure how well the prediction of PFA matches the prediction of the original RNN as fidelity (i.e., the accuracy of model extraction).

3.4 Phase 3: Apply Fault Localization to PFA

In **Phase 3**, we compute the suspiciousness scores with SBFL for sequences of state transitions on PFAs to localize the faulty ones.

We use n-grams of states (i.e., sequences of state transitions) on the PFAs to capture the behavior of the original RNNs. If n is small, the number of n-grams that PAFL considers corresponds to a small part of sequences of state transitions. On the other hand, if n is large, the number of

n-grams corresponds to a large part of the sequences of state transitions. We hypothesize that the larger number of n-grams helps grasp the prediction capability of RNNs because RNNs have numerous internal states, which are represented by sequences of states in PFAs.

When a data sample x of length T is fed into a PFA, we can obtain the sequence of PFA states $\langle s_0, \cdots, s_T \rangle$ (consisting of $T+1$ states) by the initial state s_0 and the following transitions corresponding to each of the T elements of x. From the sequence, we can obtain the consecutive n states, i.e. $\langle s_0, \cdots, s_{n-1} \rangle$, $\langle s_1, \cdots, s_n \rangle$, $\cdots$, $\langle s_{T-(n-1)}, \cdots, s_T \rangle$. Each of them is referred to as an *n-gram*. Thus, we can obtain $T - n + 2$ n-grams for a single sample x. If the original RNN fails to predict x, any of these n-grams may be responsible for the misbehavior of the RNNs. In this study, we compute the suspiciousness scores for each of these n-grams.

This setting is then formulated. X denotes a set of samples, and $u(x)$ denotes a set of n-grams from a single sample x. For example, for the case described in the previous paragraph, $u(x) = \{\langle s_0, \cdots, s_{n-1} \rangle, \langle s_1, \cdots, s_n \rangle, \cdots, \langle s_{T-(n-1)}, \cdots, s_T \rangle\}$. U denotes the collection of $u(x)$ for all the samples in X, which is formulated as follows.

$$U = \bigcup_{x \in X} u(x). \tag{3.1}$$

An example of obtaining U is shown below.

Example 3.4.1 Let $X = \{x_1, x_2, x_3, x_4\}$ be a set consisting of four samples. Assume that the sets of *bigrams* (i.e., n-grams of $n = 2$) obtained by feeding each sample into the PFA shown in Figure 3.4 are $u(x_1) = \{u_1, u_2, u_3\}$, $u(x_2) = \{u_4, u_5\}$, $u(x_3) = \{u_1, u_6\}$, and $u(x_4) = \{u_1, u_2, u_3\}$, where each u_i represents a bigram. We depict the bigrams when feeding x_1 in Figure 3.4. The state transition obtained by feeding x_1 into the PFA is $\langle S_0, S_1, S_3, S_5 \rangle$, as circled in red. We get three bigrams of PFA states: $u_1 = \langle S_0, S_1 \rangle$, $u_2 = \langle S_1, S_3 \rangle$, and $u_3 = \langle S_3, S_5 \rangle$. Therefore, the set of bigrams (i.e., n-grams) obtained by x_1 is $u(x_1) = \{u_1, u_2, u_3\}$. We obtain U by performing the same operation for the other samples (i.e., Equation 3.1). In this example, U is expressed as $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$.

In this study, we use one of the fault localization SBFL to calculate the suspiciousness score for each n-gram in U. $\mathrm{susp}(u)$ denotes the suspiciousness score of an n-gram u. We adopt five SBFL formulas shown in Table 3.1. The meanings of the variables in the formulas are as follows.

N_S: total number of data samples successfully predicted in X.

N_F: total number of data samples that failed to be predicted in X.

N_{CS}: number of successful data samples in X via the n-gram u.

Table 3.1: Studied SBFL formulae.

Name	Formula
Ample [37]	$\left\lvert \frac{N_{CF}}{N_{CF}+N_{UF}} - \frac{N_{CS}}{N_{CS}+N_{US}} \right\rvert$
Tarantula [80]	$\frac{N_{CF}/N_F}{N_{CF}/N_F+N_{CS}/N_S}$
Ochiai [7]	$\frac{N_{CF}}{\sqrt{N_F \times (N_{CS}+N_{CF})}}$
Ochiai2 [127]	$\frac{N_{CF} \times N_{US}}{\sqrt{(N_{CS}+N_{CF}) \times (N_{US}+N_{UF}) \times N_F \times N_S}}$
DStar [187]	$\frac{N_{CF}^*}{N_{CS}+N_{UF}}$

$* > 0$ is a variable of DStar. We used $* = 3$, the same as DeepFault [45] that applied the DStar to DNNs.

N_{CF}: number of failed data samples in X via the n-gram u.

N_{US}: number of successful data samples in X that do not pass through the n-gram u.

N_{UF}: number of failed data samples in X that do not pass through the n-gram u.

N_S and N_F can be obtained by prediction results of each data sample in X. N_{CS}, N_{CS}, N_{US}, and N_{UF} are computed for each n-gram based on successful and failed logs. Substituting the values of these variables into the Table 3.1 formula yields the suspiciousness scores for the n-gram. Example 3.4.2 is an example of the suspiciousness score calculation by Ochiai (i.e., $\text{susp}(u) = \text{Ochiai}(u)$).

Example 3.4.2 We assume that each sample in X in Example 3.4.1 is predicted by the extracted PFA, and x_2 and x_4 are predicted successfully, whereas x_1 and x_3 are not predicted successfully. We calculate the suspiciousness scores for each n-gram $u \in U$. The prediction result by PFA for each sample in X, obtained n-grams (i.e., bigrams) for each sample, values of N_{CS}, N_{CF}, and $\text{susp}(u)$ for each $u \in U$ are shown in Table 3.2. N_F is always two because N_F denotes the number of samples that failed to be predicted. For example, for the n-gram u_1 (i.e., a bigram in Figure 3.4), $N_{CS} = 1$ because there is only one sample x_4, which is successfully predicted via the n-gram u_1. Moreover, $N_{CF} = 2$ because there are two samples x_1 and x_3 that are not predicted successfully via n-gram u_1. Thus, we can obtain $\text{susp}(u_1) = \text{Ochiai}(u_1) = \frac{2}{\sqrt{2 \times (2+1)}} = \frac{\sqrt{6}}{3} \approx 0.82$.

The $\text{susp}(u)$ indicates the extent to which n-gram u is likely responsible for the misbehavior of the RNN. An n-gram with a high suspiciousness score indicates that the prediction of a sample is likely to fail if its transitions on the PFA include the n-gram. From the results in Table 3.2, we can say that n-gram u_1, which is associated with the highest suspiciousness score, is the most

Table 3.2: Suspiciousness scores in Example 3.4.2.

n-gram	x_1	x_2	x_3	x_4	N_{CS}	N_{CF}	susp(u)
u_1	✓		✓	✓	1	2	0.82
u_2	✓			✓	1	1	0.50
u_3	✓			✓	1	1	0.50
u_4		✓			1	0	0
u_5		✓			1	0	0
u_6			✓		0	1	0.71
res.	Fail	Suc.	Fail	Suc.			

Abbreviation: res.=prediction results

Table 3.3: ANS scores in Example 3.5.1.

sample	$u(x)$	sum of susp(u)	susp_{max}	ANS(x)
x_1	$\{u_1, u_2, u_3\}$	$0.82 + 0.50 + 0.50$	0.82	0.74
x_2	$\{u_4, u_5\}$	$0 + 0$	0	0
x_3	$\{u_1, u_6\}$	$0.82 + 0.71$	0.82	0.93
x_4	$\{u_1, u_2, u_3\}$	$0.82 + 0.50 + 0.50$	0.82	0.74

responsible n-gram for misbehavior. This indicates that the sequence of PFA states in u_1 is suspicious. Conversely, u_4 and u_5 are associated with a suspiciousness score of zero. Therefore, they do not contribute to the misbehavior.

Prior studies have reported various fault localization techniques [186], such as mutation-based, slicing-based, and learning-based fault localization techniques. We only used the most popular technique, SBFL, because our aim is to investigate the effectiveness of an existing popular fault localization technique in PAFL. Hence, investigating comprehensive fault localization techniques and clarifying the best fault localization technique on PAFL is beyond the scope of this study. However, future studies are necessary to clarify the best fault localization technique for improving the performance of PAFL.

3.5 Phase 4: Select Data

In **Phase 4**, PAFL distinguishes data samples that contribute to the misbehavior of RNNs based on faulty sequences of state transitions to evaluate the accuracy of localization. We suppose that

RNNs result in lower performance (misbehavior) on data samples that pass through localized faulty sequences. This is because these data samples are predicted using the internal states of RNNs that correspond to such faulty sequences. If RNNs result in lower performance in distinguished data samples, we would conclude that PAFL accurately localizes faulty sequences.

In this study, we calculate the *average n-gram suspiciousness score (ANS score)* for each data sample and the suspiciousness score for each n-gram. The ANS score is calculated using the suspiciousness scores of the n-grams that the data sample passed through. The ANS score can be used to select the data samples that are likely to fail to be predicted.

We now describe the calculation of ANS scores. First, for a data sample x, $u(x)$ (a set of n-grams of state transitions from x) is constructed from the state transitions obtained by feeding the sample into the PFA. ANS(x), the ANS score for sample x, is defined as the average of the suspiciousness scores of n-grams in $u(x)$:

$$\text{ANS}(x) = \frac{1}{|u(x)|} \sum_{u_i \in u(x)} \frac{\text{susp}(u_i)}{\text{susp}_{max}}. \tag{3.2}$$

In the equation above, $\text{susp}_{max} = \max_{u_i \in u(x)} \text{susp}(u_i)$ is the maximum value among the suspiciousness scores of n-grams in $u(x)$. By dividing the suspiciousness score by the maximum value, the maximum suspiciousness score of each n-gram in $u(x)$ is normalized to one. $|u(x)|$ indicates the number of n-grams of $u(x)$. Example 3.5.1 shows an example of ANS score calculation.

Example 3.5.1 We suppose that the set X consists of the four data samples we addressed in Examples 3.4.1 and 3.4.2. The ANS score for each sample of X can be calculated based on the suspiciousness scores for the n-grams calculated in Example 3.4.2. The ANS scores for the samples of X are shown in Table 3.3. For example, for x_1, the bigrams (i.e., n-grams of $n = 2$) obtained by feeding x_1 to PFA are u_1, u_2, and u_3 ($|u(x_1)| = 3$). Additionally, the maximum suspiciousness score among u_1, u_2, and u_3 is 0.82, which is the suspiciousness score of u_1 (i.e., $\text{susp}_{max} = 0.82$). Thus, the ANS score of the sample x_1 is as follows: $\text{ANS}(x_1) = \frac{1}{3}\{\frac{1}{0.82}(0.82+0.50+0.50)\} \approx 0.74$. We depict ANS$(x_1)$ in Figure 3.4. Finally, we select data samples based on ANS scores. In this example, we may select x_3, which has the highest ANS score.

4 Experimental Setup

This section provides an overview of the experiment. Specifically, we explain the research questions, an overview of the experiment, datasets, model settings, evaluation criteria, and comparison with the state-of-the-art fault localization for RNNs.

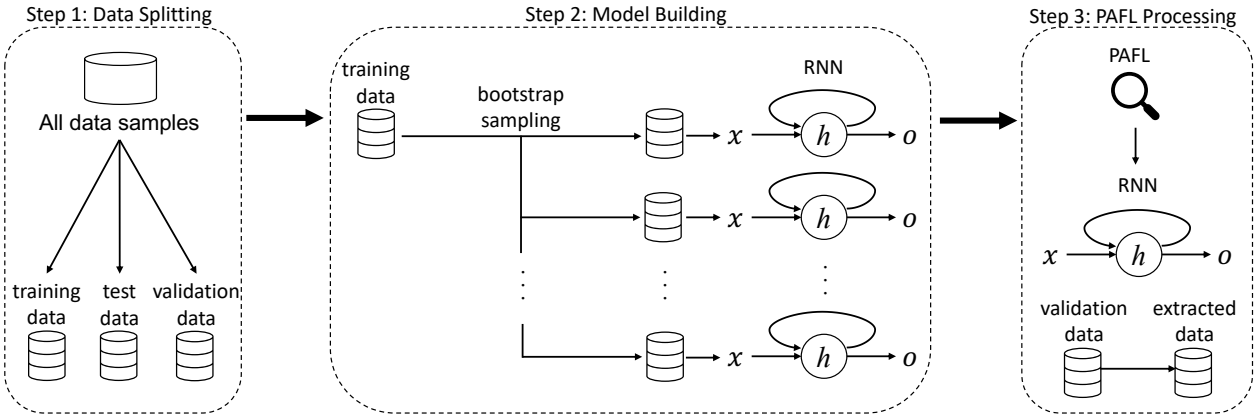


Figure 3.5: Overview of our experiment.

4.1 Research Questions

We prepare the following two research questions to evaluate PAFL.

RQ1: Do ANS scores represent the misbehavior of RNNs?

In PAFL, the ANS score is used to select data samples related to the misbehavior of RNNs. The score should represent the misbehavior of RNNs to effectively select data samples. In this RQ, we investigate whether the distribution of the ANS scores per dataset represents the misbehavior of the model corresponding to the dataset.

RQ2: Does PAFL identify data samples that cause the misbehavior of RNNs?

The results of RQ1 indicates whether the ANS score represents the misbehavior of RNNs. However, they do not reveal whether the data samples selected based on the ANS score actually cause a failure in the prediction. In this RQ, we investigate whether PAFL can select data samples where the RNN is incapable of accurate prediction.

4.2 Overview

In this section, we provide an experimental overview (Figure 3.5), followed by a detailed description of each step.

Step 1: **Dataset Splitting.** We randomly split the data samples in the dataset into training, validation, and test data. Training data is used to train RNNs, validation data is used to apply PAFL, and test data is used to evaluate trained models.

Step 2: **Model Building.** In the experiment, we use three popular variants of RNNs: simple recurrent neural networks (SRNNs), gated recurrent units (GRUs), and long short-term memories (LSTMs). SRNNs have a simple architecture and are refered to an Elman net-

work [44]. GRUs [31] and LSTMs [156] have more complex architectures than SRNNs because of their gating mechanisms. Additionally, bootstrap sampling is performed on the training data for each dataset multiple times, which provides multiple trained models for each dataset. This enables us to perform statistical tests of the prediction performance of the model.

Step 3: **PAFL Processing.** We apply PAFL to each model trained with bootstrap samples and select data samples from the validation data, based on the ANS score. Notably, multiple PFAs can be extracted for a single dataset because of the multiple bootstrap samplings and variation in the value of the number of clusters k in the PFA construction.

We build RNN models and implement PAFL using Python, PyTorch [1], and scikit-learn [146]. Specifically, we build SRNNs, GRUs, and LSTMs using `torch.nn.RNN`, `torch.nn.GRU`, and `torch.nn.LSTM` in PyTorch API, respectively.

4.3 Datasets

We use eight datasets, which were used in previous studies for the experiments [40, 194]. MNIST and TOXIC are the datasets used in a prior study, which proposes an automatic RNN repair approach called RNNRepair [194], and the remaining six datasets are used in the study for extracting PFAs from RNNs [40]. The seven datasets, excluding MNIST, are binary classification datasets. We use a quarter of the training data as the validation data and the remainder is used for training. We show the number of data samples for the training, validation, and test data for each dataset in Table 3.4.

Tomita Grammars. Tomita Grammars are one of the most popular benchmark datasets for this purpose. These datasets include seven grammatical rules for strings on $\{0, 1\}$. Each corresponds to a dataset from Tomita1 to Tomita7. Each data sample in the dataset of Tomita Grammars is a string of zeros or ones. Each data sample is labeled 1 (positive) if it satisfies the rule and 0 (negative) if otherwise. We create the datasets using the same program implemented in [40]. We use only Tomita3, 4, and 7 because Tomita1, 2, 5, and 6 are not used for prediction using PFA, owing to label imbalance. We convert each data sample into a two-dimensional one-hot vector, because the sample consists of a string of $\{0, 1\}$.

BP [184]. This is an artificial dataset consisting of 28 different characters (i.e., “a” - “z, ” “ (,” “)”). Each string is labeled 1 (positive) if the parentheses are correctly closed and 0 (negative) if otherwise. We create a BP dataset using the program implemented in [40]. The dataset contains 3,980 and 995 data samples from the original training and test data, respectively. We split the

original training data into 2,985 and 995 data samples for training and validation, respectively. Each character is converted into a 28-dimensional one-hot vector to convert strings into numerical vectors because there are 28 different characters.

RTMR [142]. This movie-review dataset is used for sentiment analysis. Sentiment analysis determines whether the content of each review sentence is positive or negative. The content of the review sentence is labeled 1 if it is positive and 0 if otherwise. This dataset contains 8,530 and 2,132 data samples from the original training and test data, respectively. We split the original training data into 6,397 and 2,132 data samples for training and validation, respectively. We use a word2vec model pretrained on the Google News corpus [121] to convert words into a 300-dimensional vector.

IMDB [112]. This is a dataset of movie reviews used for sentiment analysis, similar to RTMR. IMDB is originally a large dataset with 25,000 training and test datasets. However, in [40], reviews consisting of less than or equal to 50 words are used, and the experiments follow this setting. Consequently, 3,730 and 933 samples remained for the training and test data, respectively. We split the training data into 2,798 and 932 data samples for training and validation, respectively. We use the word2vec model that is used for RTMR to convert the words into a 300-dimensional vector.

TOXIC [2]. This is a natural language dataset available on Kaggle [2]. Each sample consists of a comment on a Wikipedia talk page labeled toxic or not. Toxic comments are rude and abusive to others. Originally, it contains 18,968 training and 2,108 test samples. We divide the original training sample into 14,226 training and 4,742 validation samples. Each word in the sample is converted into a 300-dimensional vector by the same word2vec model used for RTMR and IMDB.

MNIST [101]. MNIST is a handwritten digit classification dataset. Each sample is a 28x28 gray-scale image and has a label indicating the digit. MNIST originally consists of 60,000 training samples and 15,000 test samples. We divide the original training samples into 45,000 training and 15,000 validation samples. This dataset differs from the other seven datasets in this study in the following ways: (1) it has ten classes and (2) it consists of images instead of texts. We consider an image a sentence (a sample) and each row (i.e., 28-pixel values) of the image a word in the other datasets. RNNs process each row from the top.

4.4 Model Settings

We applied the approach PAFL to three RNN variants, SRNNs, GRUs, and LSTMs, to evaluate PAFL in various RNNs. To build these RNNs, we have to determine their parameters: *the dimension of the hidden state*, *the number of hidden layers*, and *the number of training epochs*. As our aim is not

Table 3.4: Number of data samples for each dataset.

Dataset	#training data	#validation data	#test data
Tomita3	2,174	724	724
Tomita4	2,183	727	727
Tomita7	2,159	719	719
BP	2,985	995	995
RTMR	6,397	2,132	2,133
IMDB	2,798	932	933
TOXIC	14,226	4,742	2,108
MNIST	45,000	15,000	10,000

to accomplish the highest performance, we used the parameters used in prior studies [40, 194]. Specifically, we used two parameter settings because the studied datasets were collected from these studies. When using the Tomita3, 4, 7, BP, RTMR, and IMDB datasets, we followed the setting of [40]. Specifically, the dimension of the hidden states is 512, the number of hidden layers is one, and the number of epochs is 100. When using the MNIST and TOXIC datasets, we followed the setting of [194]. Specifically, the dimension of the hidden states is 100 for MNIST and 300 for TOXIC, the number of hidden layers is one, and the number of epochs is 15.

Bootstrap sampling was used to perform statistical tests on the performance of PAFL. For each dataset, bootstrap sampling randomly samples the data with replacement from the training data, and a set of samples is used as the training data (i.e., the bootstrap sample). The number of samples extracted from one bootstrap sample is equal to the number of samples in the training data. For each dataset, this bootstrap sampling was repeated ten times, and the model was trained using each of them. Therefore, PAFL can be applied to ten RNNs for each dataset for each RNN variant. For each dataset, the results of predicting the test data with each of the ten models and their aggregation are shown in Table 3.5. Specifically, we computed the prediction labels and evaluation criteria (described in the next section) for each of the ten models, and averaged these values across all models.

4.5 Evaluation Criteria

We evaluated PAFL with the prediction performance of RNNs because PAFL should be used to evaluate the misbehavior of RNNs. We evaluated the prediction performance of RNNs using multiple evaluation criteria. This is because one evaluation criterion may be good and the other may

be bad. For example, improving the precision may decrease the recall. Specifically, we used six evaluation criteria: accuracy (acc.), area under the receiver operating characteristic curve (AUC), f1-score (F1), Matthews correlation coefficient (MCC), precision (pre.), and recall (rec.). AUC is a threshold-independent criterion, whereas the others are threshold-dependent criteria. We used 0.5 as the threshold to compute the threshold-dependent criteria. Except for MCC, the range of all criteria is $[0, 1]$, and the higher the value, the better is the performance. For MCC, the range is $[-1, 1]$, and the higher the absolute value, the better is the performance. As the task in the MNIST dataset is not binary classification but ten-class classification, the calculation of the evaluation criteria is slightly different. Specifically, for pre., rec., F1, and AUC, a macro average, which calculates the criteria for each class and averages them, was taken.

Moreover, we conducted hypothesis testing and effect size calculation for RQ2. We used the Mann–Whitney U test [114], which is a nonparametric test. We used Cliff's delta [33], an effect size in addition to the p-value for the test to evaluate the degree of difference. Cliff's delta d can be calculated using the test statistic U as follows.

$$d = \frac{2U}{n_1 n_2} - 1 \tag{3.3}$$

where n_1 and n_2 denote the sample sizes of each group. Cliff's delta takes values between -1 and 1, with the absolute value indicating the degree of difference in the distribution, and the sign indicating the larger group.

4.6 Comparison with RNNRepair

For RQ2, we investigate whether PAFL is superior to a state-of-the-art FL technique. For this investigation, we used the FL part of RNNRepair, a state-of-the-art automated RNN repair technique [194]. We considered the FL part of RNNRepair as a state-of-the-art FL technique for RNNs because this study even proposes and evaluates repairing techniques for RNNs. This study proposed two fault localization techniques: *sample-level* and *segment-level* ones. The former detects faulty samples, and the latter detects faulty segments (i.e., elements that comprise a sample). Because PAFL is a sample-level fault localization technique, we compared PAFL with the sample-level fault localization proposed by [194]. We call it *SOTA FL (state-of-the-art fault localization)*. SOTA FL detects samples that highly influence misbehavior in the test data by using *influence analysis*, which computes the degree of influence between samples.

Table 3.5: Prediction performance of trained models.

Model	Dataset	acc.	pre.	rec.	F1	MCC	AUC
SRNN	Tomita3	0.927	0.685	0.840	0.753	0.717	0.963
	Tomita4	0.950	0.921	0.927	0.923	0.887	0.986
	Tomita7	0.978	0.857	0.906	0.878	0.868	0.995
	BP	0.573	0.579	0.590	0.584	0.145	0.565
	RTMR	0.764	0.806	0.690	0.740	0.535	0.837
	IMDB	0.823	0.839	0.846	0.842	0.642	0.899
	TOXIC	0.869	0.867	0.866	0.866	0.854	0.984
	MNIST	0.881	0.857	0.916	0.885	0.765	0.951
GRU	Tomita3	0.987	0.871	0.983	0.923	0.913	0.996
	Tomita4	0.970	0.918	0.996	0.956	0.935	0.992
	Tomita7	0.997	0.985	0.983	0.983	0.982	1.000
	BP	0.992	0.990	0.994	0.992	0.984	0.997
	RTMR	0.774	0.797	0.731	0.760	0.552	0.845
	IMDB	0.826	0.838	0.853	0.845	0.646	0.900
	TOXIC	0.903	0.885	0.927	0.906	0.807	0.960
	MNIST	0.919	0.918	0.918	0.918	0.910	0.994
LSTM	Tomita3	0.994	0.957	0.997	0.977	0.973	0.999
	Tomita4	0.974	0.942	0.982	0.961	0.942	0.991
	Tomita7	0.996	0.986	0.970	0.978	0.976	1.000
	BP	0.988	0.983	0.993	0.988	0.976	0.996
	RTMR	0.771	0.790	0.738	0.759	0.547	0.843
	IMDB	0.827	0.836	0.860	0.847	0.649	0.898
	TOXIC	0.896	0.882	0.915	0.898	0.793	0.954
	MNIST	0.942	0.942	0.942	0.942	0.936	0.997

5 Experimental Results

5.1 RQ1) Do ANS scores represent the misbehavior of RNNs?

Approach. In this RQ, we compare the average ANS scores for each bootstrap sample between datasets. We compute and average the ANS scores of all the data samples in a bootstrap sample.

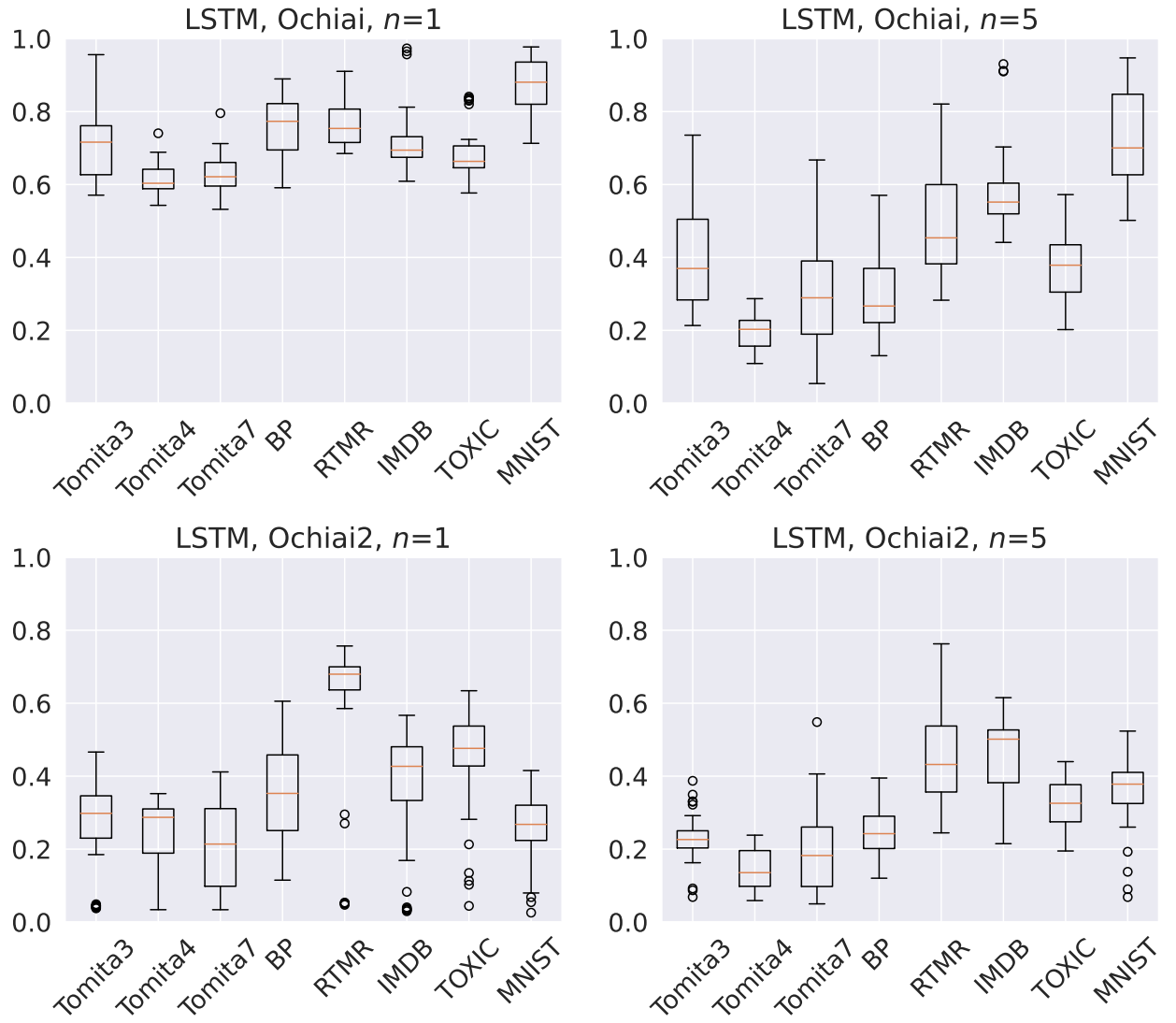


Figure 3.6: Average ANS scores for each n and dataset.

We investigate whether the average ANS scores for bootstrap samples depend on the prediction results of the test data (Table 3.5) and clarify whether ANS scores represent the misbehavior of RNNs. The candidates of parameters of PAFL for computation of ANS scores are as follows. The number of clusters (i.e., k) for PFA is two, four, six, eight, and ten, and the number of n-grams (i.e., n) for calculating the suspiciousness score is one, two, three, four, and five. To compute the suspiciousness scores for an n-gram, we use the five SBFL formulas shown in Table 3.1.

Results. Figure 3.6 shows the box plot of ANS scores for each dataset for $n = 1, 5$. Each box plot consists of 50 average ANS scores from five k and ten bootstrap samples (5×10). The figure shows the results when the variant is LSTM and the formulas are Ochiai and Ochiai2. The other settings are omitted due to space limitations, but similar trends are observed, and all results can

be found in our replication package.

When $n = 5$, the datasets in which RNNs result in worse performance (i.e., RTMR, IMDB, TOXIC, and MNIST) have higher median ANS scores (approximately 0.4–0.6) than the other four datasets (approximately 0.2–0.4). Table 3.5 shows that LSTMs achieve a worse performance for the RTMR, IMDB, TOXIC, and MNIST datasets than for the other datasets. We call these datasets *difficult datasets*. LSTMs using such difficult datasets should misbehave more compared to LSTMs using other datasets. Hence, ANS scores would be higher for these datasets if ANS scores represented the misbehavior of LSTMs. Figure 3.6 shows that median ANS scores for these datasets are higher than those for the other datasets when $n = 5$, regardless of using either Ochiai or Ochiai2. Therefore, the ANS score can capture the misbehavior of LSTMs when $n = 5$.

When $n = 1$, ANS scores do not represent differences in the prediction performance of the models corresponding to the dataset compared to the case $n = 5$. When $n = 1$ in Figure 3.6, ANS scores are inconsistent with the prediction performance of LSTMs. For example, the median ANS score for all datasets except MNIST is approximately 0.6–0.8 when using Ochiai. However, Table 3.5 shows that LSTMs achieve a worse performance for the difficult datasets (i.e., RTMR, IMDB, TOXIC, and MNIST). Hence, $n = 1$ overlooks the differences in the prediction performance across the studied datasets.

The results for RQ1 imply that the idea of considering a sequence of states by using n-grams is important for representing the misbehavior of RNNs using the ANS score, as described in Section 3.4.

> **Summary of RQ1:** ANS scores with sequences of states represent the misbehavior of RNNs. Indeed, the medians of the ANS scores for the RTMR, IMDB, TOXIC, and MNIST datasets show higher values than those of the other datasets except when $n = 1$.

5.2 RQ2) Does PAFL identify data samples that cause the misbehavior of RNNs?

Approach. Figure 3.7 shows the experimental procedure for RQ2. In this experiment, we compare the prediction performance of RNNs on selected data samples by PAFL (*PAFL-selected data*) with the performance on randomly selected data samples (*randomly selected data*) and data samples selected by SOTA FL (*SOTA FL-selected data*) in the validation dataset. The validation data are used to prevent data leakage. If the prediction performance on PAFL-selected data is worse than that of randomly selected data or SOTA FL-selected data, we would conclude that PAFL can distinguish data samples that cause misbehavior. The data selection process using PAFL is as follows:

1) input each data sample in the validation dataset into the PFA and calculate the ANS score

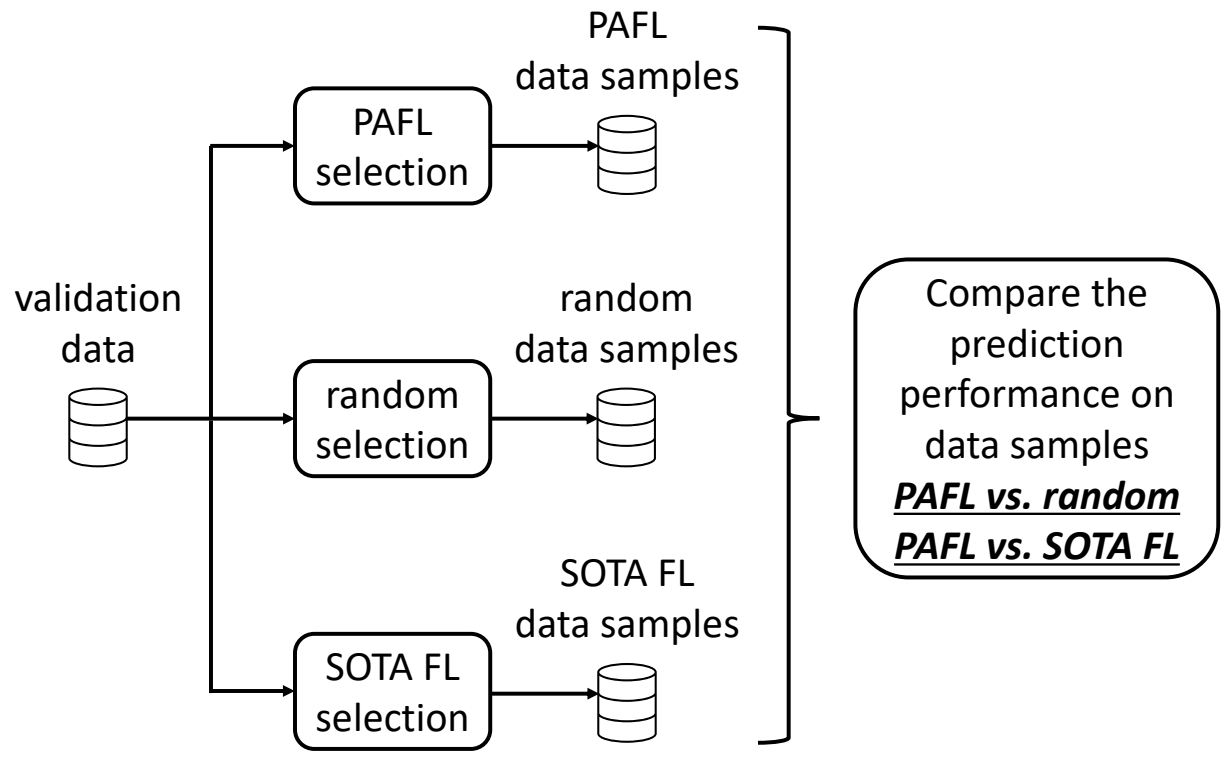


Figure 3.7: Overview for RQ2 approach.

for each data sample,

2) select data samples with the top θ% of ANS scores as the PAFL-selected data.

The value of θ is a parameter, and we use 10 and 50 as the candidate parameters. We use the same parameter candidates for k and n used in the RQ1 experiment. We study all the combinations of these parameter candidates (*parameter settings*), the three variants, and the five SBFL formulas. When comparing with SOTA FL, we use the fixed parameters of SOTA FL for each dataset, which were well explored in the appendix of [194]. SOTA FL-selected data with those parameters are compared to PAFL-selected data with the best parameters (k, n) for each dataset and variant. For each setting, we compute the average prediction performance for each model trained by ten bootstrap samples on PAFL-selected, randomly selected, or SOTA FL-selected data, and compare them.

Results. Tables 3.6 and 3.7 show the percentage of settings where the average prediction performance of RNNs trained by ten bootstrap samples on PAFL-selected data is worse than that on randomly selected data. Table 3.6 shows the percentages for each RNN variant and dataset, and Table 3.7 shows the percentages for each RNN variant and SBFL formula. The value in paren-

Table 3.6: The percentage of settings where the average prediction performance on PAFL-selected data is worse than that on randomly selected data for each dataset.

Variant	Tomita3	Tomtia4	Tomita7	BP	**Avg.**
SRNN	70.60% (28.07%)	76.47% (22.20%)	53.13% (14.47%)	47.33% (7.93%)	**61.88% (18.17%)**
GRU	55.53% (24.87%)	78.27% (19.93%)	54.60% (5.73%)	74.20% (10.40%)	**65.65% (15.23%)**
LSTM	65.80% (8.27%)	79.67% (27.27%)	62.87% (12.47%)	51.73% (3.00%)	**65.02% (12.75%)**
Variant	RTMR	IMDB	TOXIC	MNIST	**Avg.**
SRNN	90.67% (72.80%)	86.93% (78.67%)	89.53% (80.80%)	84.80% (45.93%)	**87.98% (69.55%)**
GRU	93.33% (74.93%)	88.80% (74.27%)	91.47% (80.87%)	87.07% (61.13%)	**90.17% (72.80%)**
LSTM	93.47% (69.33%)	86.33% (73.27%)	91.07% (79.87%)	87.53% (63.47%)	**89.60% (71.48%)**

In parentheses shows the percentage of indicating statistically significant differences and large effect size (Table 3.7 is similar).

Table 3.7: The percentage of settings where the average prediction performance on PAFL-selected data is worse than that on randomly selected data for each SBFL formula.

Variant	Ample	Tarantula	Ochiai	Ochiai2	DStar	**Avg.**
SRNN	53.20% (21.33%)	86.88% (54.96%)	77.83% (49.13%)	83.38% (50.63%)	73.38% (43.25%)	**74.93% (43.86%)**
GRU	61.42% (21.46%)	84.63% (55.58%)	82.08% (49.71%)	83.21% (49.67%)	78.21% (43.67%)	**77.91% (44.02%)**
LSTM	60.25% (21.46%)	88.71% (52.96%)	79.25% (46.58%)	88.58% (45.96%)	69.75% (43.63%)	**77.31% (42.12%)**

theses is the percentage of the cases where the U test between prediction performance on PAFL-selected and randomly selected data indicates $p < 0.01$ and the effect size is positive and large (i.e., ≥ 0.474 [153]). The population for calculating percentages is 1,500 in Table 3.6 (five $k \times$ five $n \times$ two $\theta \times$ six metrics $\times$ five formulas) and 2,400 in Table 3.7 (five $k \times$ five $n \times$ two $\theta \times$ six metrics $\times$ eight datasets).

For all RNN variants, PAFL successfully selects data samples that result in statistically significantly lower prediction performance with large effect sizes from 69.55% to 72.80% for the RTMR, IMDB, TOXIC, and MNIST datasets. We observed this result for the Table 3.6. It is worth noting that, for the Tomita3, Tomita4, Tomita7, and BP datasets, almost all prediction results on PAFL-selected data are still worse than those on randomly selected data (61.88% to 65.65%); however, many cases do not have statistically significant differences with large effect sizes (12.75% to 18.17%). One reason is the difficulty of the datasets. Table 3.5 shows that, generally, RNNs result in higher performance for the Tomita3, Tomita4, Tomita7, and BP datasets than for the other datasets; therefore, few samples cause misbehavior. In summary, PAFL-selected data samples are

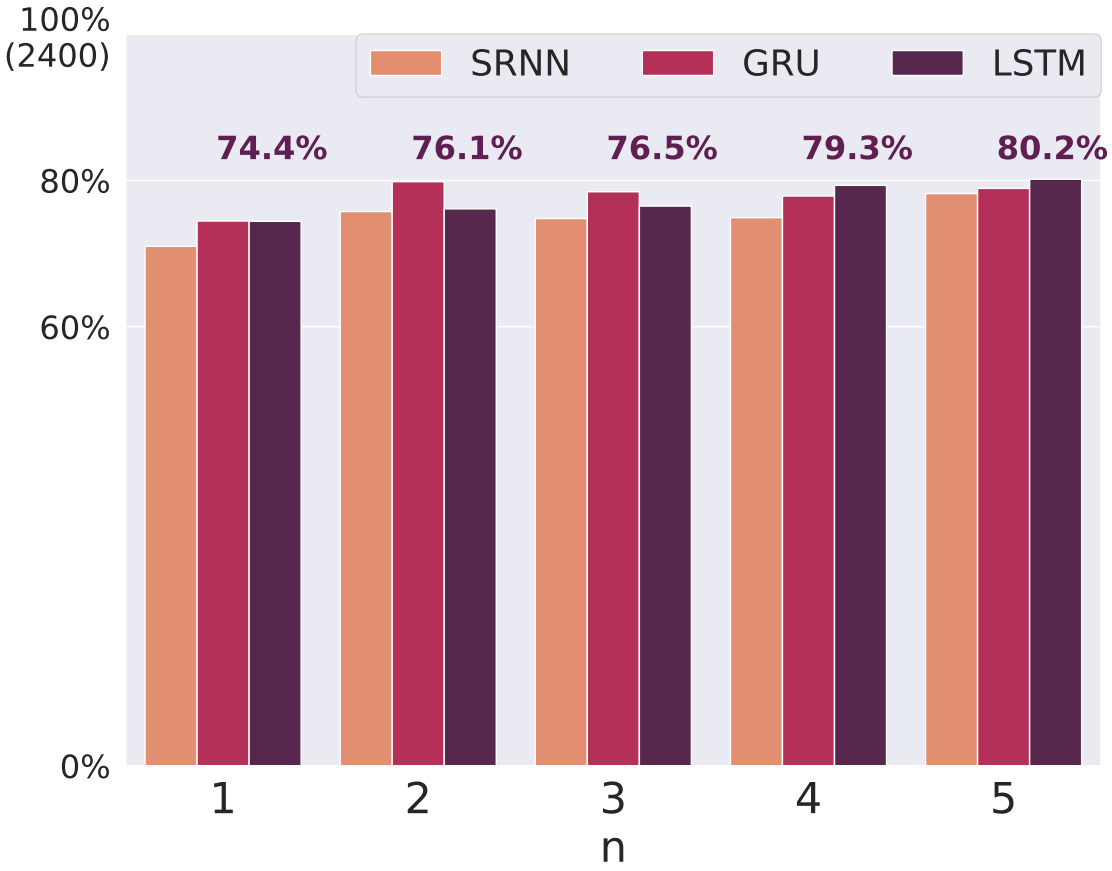


Figure 3.8: Numbers of parameter settings in which PAFL-selected data lead to worse performance for each n.

more likely to result in misbehavior compared to randomly selected data samples for all RNN variants, especially for datasets where RNNs result in worse performance.

In terms of the SBFL formula, Tarantula gave the best results (84.63% to 88.71% on average) for all variants. Table 3.7 shows that Ochiai2 is close behind Tarantula for all variants, and it results in lower prediction performance on average (more than 80%) for any variants. The rank of the formula is as follows: Tarantula, Ochiai2, Ochiai, DStar, and Ample. PAFL is effective no matter which formula is used as even Ample can select data with low prediction performance in a majority of cases (at least 53.20%).

For LSTMs, PAFL effectively distinguishes data samples more when $n = 5$ than for lower n. Figure 3.8 shows the number of parameter settings for which PAFL-selected data result in worse performance than randomly selected data for each n. The maximum number is 2,400 (five k $\times$ eight datasets $\times$ two θ $\times$ six evaluation criteria $\times$ five formulas). We observe that the larger the value of n, the larger the number. This shows that n is dominant for PAFL performance

Table 3.8: The percentage of settings where the average prediction performance on PAFL-selected data is worse than that on SOTA FL-selected data for each dataset.

Variant	Tomita3	Tomtia4	Tomita7	BP	**Avg.**
SRNN	33.33% (36.67%)	63.33% (56.67%)	93.33% (86.67%)	20% (28.33%)	**52.50% (47.50%)**
GRU	91.67% (78.33%)	88.33% (36.67%)	91.67% (56.67%)	96.67% (33.33%)	**92.09% (51.25%)**
LSTM	100% (86.67%)	98.33% (46.67%)	100% (33.33%)	100% (23.33%)	**99.58% (52.08%)**
Variant	RTMR	IMDB	TOXIC	MNIST	**Avg.**
SRNN	98.33% (73.33%)	93.33% (53.33%)	100% (91.67%)	91.67% (83.33%)	**94.17% (75.42%)**
GRU	100% (81.67%)	91.67% (75.00%)	100% (100%)	91.67% (90.00%)	**96.00% (86.67%)**
LSTM	100% (80.00%)	76.67% (83.33%)	100% (100%)	100% (86.67%)	**95.83% (87.50%)**

In parentheses shows the percentage of indicating statistically significant differences and large effect size (Table 3.9 is similar).

Table 3.9: The percentage of settings where the average prediction performance on PAFL-selected data is worse than that on SOTA FL-selected data for each SBFL formula.

Variant	Ample	Tarantula	Ochiai	Ochiai2	DStar	**Avg.**
SRNN	71.88% (62.50%)	75.00% (64.58%)	73.96% (58.33%)	79.17% (68.75%)	70.83% (64.58%)	**74.17% (67.50%)**
GRU	92.71% (68.75%)	94.79% (69.79%)	93.75% (65.63%)	94.79% (72.92%)	93.75% (67.71%)	**93.96% (68.96%)**
LSTM	95.83% (63.54%)	97.92% (71.88%)	95.83% (61.46%)	97.92% (80.21%)	96.88% (60.42%)	**96.88% (63.75%)**

improvement compared to other parameters (i.e., k, θ, and SBFL formulas). For LSTMs, especially, the percentage increases monotonically with n (74.4%, 76.1%, 76.5%, 79.3%, and 80.2% for n of 1, 2, 3, 4, and 5, respectively), but this is not the case for GRUs and SRNNs. As LSTM has a more complex architecture than a GRU and SRNN, increasing n to capture the behavior in detail seems to work effectively as described in Section 3.4.

Tables 3.8 and 3.9 show the percentage of settings where the prediction performance of models on PAFL-selected data is worse compared to that on SOTA FL-selected data. Similar to Tables 3.6 and 3.7, the value in parentheses is the percentage of the cases where the U test between prediction performance on PAFL-selected and SOTA FL-selected data indicates $p < 0.01$, and the effect size is positive and large (i.e., ≥ 0.474 [153]). The population for calculating percentages is 60 in Table 3.8 (two θ $\times$ six metrics $\times$ five formulas) and 96 in Table 3.9 (two θ $\times$ six metrics $\times$ eight datasets). The number of populations is smaller than in Tables 3.6 and 3.7 because we are comparing the fixed parameter case of SOTA FL with the best parameter case of PAFL.

For all RNN variants, PAFL successfully selects data samples that result in statistically

significantly lower prediction performance with large effect sizes from 75.42% to 87.50% for the RTMR, IMDB, TOXIC, and MNIST datasets. This is a similar result to the comparison with the random-selected data, but Table 3.8 shows that PAFL-selected data samples are more likely to result in misbehavior compared to SOTA FL-selected data samples for all RNN variants, especially for the difficult datasets. Note that the result of the SRNN for the Tomita3, 4, and 7, and BP datasets does not indicate that over half of the cases result in statistically significant differences; however, over half of the cases exhibit a worse performance except for the Tomita3 and BP datasets.

In terms of the SBFL formula, Ochiai2 gave the best results (79.17% to 97.92% on average) for all variants. Table 3.9 shows that Tarantula is close behind Ochiai2 or the same for all variants. Hence, we observe similar but slightly different results from the comparison with randomly selected samples. In summary, the results imply that both Tarantula and Ochiai2 seem to be superior to the other formulas.

These results imply that ANS scores are superior to the parameter that is used in SOTA FL. SOTA FL uses the *influence scores*, which are defined by the similarity between data samples [194]. The results of RQ2 reveal that PAFL is superior to SOTA FL; hence, we conclude that ANS scores are also superior to the influence scores in terms of selecting faulty data samples.

Summary of RQ2: PAFL better distinguishes data samples that cause the misbehavior of RNNs compared to random selection and SOTA FL. PAFL, in particular, performs well with difficult datasets. In fact, PAFL statistically significantly outperforms SOTA FL in 75.42%–87.50% of all cases for the RTMR, IMDB, TOXIC, and MNIST datasets. Additionally, in terms of the SBFL formula, Tarantula and Ochiai2 likely show better performances than the other formulas.

6 Discussion

6.1 Is PAFL superior to SOTA FL in terms of execution time?

Motivation and Approach. In RQ2, we compared PAFL and SOTA FL in terms of the performance of selecting data samples that cause misbehavior and showed PAFL is superior to SOTA FL. While this result encourages developers who build RNNs to use PAFL to localize faulty data samples, we do not show the execution cost of using PAFL in RQ2. If the execution cost of using PAFL is significantly higher than SOTA FL and impractical, PAFL may not be used in development.

In this discussion, we evaluate the execution cost of using PAFL and compare it with that of SOTA FL. Specifically, we consider two phases, *the abstraction phase* and *the data selection phase*,

Table 3.10: Execution time for PAFL and SOTA FL.

Dataset	Total [sec.]		Abstraction [sec.]		Data selection [sec.]	
	PAFL	SOTA FL	PAFL	SOTA FL	PAFL	SOTA FL
Tomita3	4.30	3.24	4.29	3.06	0.01	0.18
Tomita4	4.37	4.31	4.36	4.12	0.01	0.19
Tomita7	4.32	4.45	4.32	4.31	0.01	0.14
BP	7.15	4.17	7.14	3.56	0.01	0.61
RTMR	56.82	18.92	56.80	14.56	0.02	4.36
IMDB	247.48	11.99	247.47	11.19	0.01	0.80
TOXIC	200.04	20.51	199.99	15.36	0.05	5.00
MNIST	163.39	91.62	163.18	31.82	0.21	59.80

to discuss not only the entire execution time but also the execution time of preparation of fault localization (i.e., the abstraction phase) and fault localization itself (i.e., the data selection phase). In PAFL, the abstraction phase indicates extracting PFAs from RNNs; the data selection phase indicates computing ANS scores to select data samples. SOTA FL applies GMM (Gaussian Mixture Model)[3] to hidden states to classify hidden states; we consider this operation the abstraction phase. Furthermore, SOTA FL uses influence analysis to compute the influence scores to select data samples that highly affect misbehavior [194]; we consider this operation the data selection phase.

Result. The maximum execution time of PAFL is around 4 min, and PAFL results in a lower execution time in the data selection phase than SOTA FL. Table 3.10 shows the average execution time across all bootstrap samples and all parameter combinations. We observe that the total execution times are not so different between PAFL and SOTA FL for the datasets in which RNN results in high prediction performance (i.e., Tomita3, 4, 7, and BP). However, PAFL and SOTA FL exhibit differences for other datasets that are difficult, as described in Section 5.1. These differences are caused by the abstraction phase. While the maximum average execution time of SOTA FL in the abstraction phase is 31.82 s, that of PAFL is 247.47 s. However, this execution time is around 4 min, and once PFAs are extracted, they do not need to be extracted again. Furthermore, the average execution time of the data selection phase of PAFL is lower than that of SOTA FL. In summary, while PAFL takes a longer execution time than SOTA FL, PAFL is a practical approach to localizing faulty data samples.

[3] https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html

Summary of Discussion1: While PAFL takes a longer execution time than SOTA FL, it is a practical approach for localizing faulty data samples. Indeed, even the maximum average execution time is around 4 min. Because the abstraction phase consumes almost all of the computation time and this phase does not need to be executed multiple times, PAFL usually needs less than a second to select data samples.

6.2 Do PAFL parameters need to be optimized?

Motivation and Approach. The results of RQ2 reveal that PAFL can distinguish data samples that cause the misbehavior of RNNs. However, we do not examine the optimization of parameters. PAFL has two important parameters: the number of clusters k of a PFA and n of n-grams. Specifically, n may affect the performance of PAFL given the results for RQ1 and RQ2. In this discussion, we investigate whether we have to optimize the parameters (k, n) for each dataset when using PAFL.

To clarify whether the parameters must be optimized, we investigate the combination of k and n in which the RNN presents the lowest/highest prediction performance on PAFL-selected data samples for each bootstrap sample of datasets. If (k, n) is consistent across bootstrap samples, then we can select (k, n) without any optimization. Otherwise, we must optimize (k, n) for each dataset.

Result. Figure 3.9 shows the distribution of parameter values for accuracy. The figure shows the results when the variant is LSTM or SRNN, and the formula is Ochiai or Ample. The other settings are omitted due to space limitations, but similar trends are observed, and all results can be found in our replication package. The x-axis and y-axis show the values of parameters k and n, respectively. Each dot represents the combination of (k, n) that leads to the lowest or highest prediction performance among all combinations of (k, n) for each bootstrap sample of the datasets. For example, in the plot with the lowest accuracy in LSTM with Ochiai (upper left), we observe a dot at $(2, 1)$. This dot indicates one or more bootstrap samples from which PAFL with the parameter $(2, 1)$ retrieves data samples, where LSTM results in the lowest accuracy.

The parameter values that lead to the lowest or highest prediction performance vary across target bootstrap samples, RNN variants, and SBFL formulas. Generally, we do not observe consistent parameter values where PAFL distinguished the data samples that led to the highest or lowest prediction performance (Figure 3.9). For example, in LSTM with Ochiai, PAFL may work effectively with $(k, n) = (2, 1)$; however, it may also work effectively with $(k, n) = (10, 5)$. Similarly, we observed that (k, n) for the lowest or highest prediction performance depends on

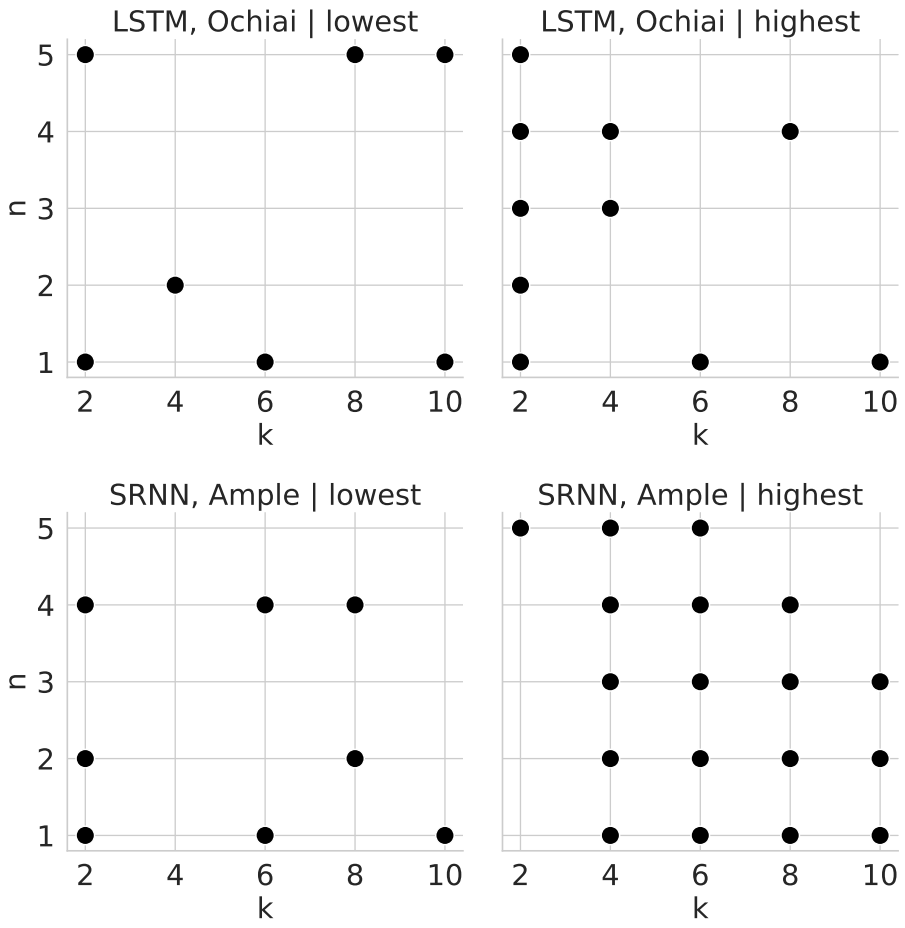


Figure 3.9: Parameters (k, n) at the lowest/highest in terms of accuracy in the selected data samples.

the bootstrap samples, RNN variants, and the SBFL formulas. Hence, the best parameter value may depend on all factors; developers who use PAFL must determine the parameter values in an exploratory manner, such as optimizing hyperparameters in machine learning models. For example, developers would search for (k, n) such that PAFL works effectively using the validation data and use the (k, n) in the test data.

Summary of Discussion2: Yes, users must optimize the parameters of PAFL. The results for RQ2 demonstrate that n probably changes the performance of PAFL; however, several exceptions exist. Future studies are necessary to reveal the impact of the parameters of PAFL. We recommend that users who use PAFL optimize the parameters k and n similar to optimizing the hyperparameters of machine learning models.

Table 3.11: Percentages of data samples in critical ANS ties. Bolded percentages indicate k that leads to the fewest critical ANS ties for each variant and dataset.

Variant	Dataset	$k = 2$	$k = 4$	$k = 6$	$k = 8$	$k = 10$
SRNN	Tomita3	46.13%	23.70%	8.29%	5.64%	**4.15%**
	Tomita4	10.36%	1.40%	0.86%	0.72%	**0.58%**
	Tomita7	3.47%	1.39%	0.65%	**0.51%**	**0.51%**
	BP	55.54%	27.02%	21.09%	17.42%	**15.99%**
	RTMR	68.68%	37.57%	21.34%	11.99%	**7.63%**
	IMDB	16.25%	0.53%	0.04%	**0.00%**	0.02%
	TOXIC	65.08%	26.45%	15.77%	10.80%	**8.66%**
	MNIST	96.57%	69.25%	47.47%	34.03%	**24.74%**
GRU	Tomita3	24.35%	1.15%	0.47%	0.20%	**0.12%**
	Tomita4	16.10%	1.34%	0.22%	0.21%	**0.18%**
	Tomita7	6.71%	0.43%	**0.09%**	0.80%	3.02%
	BP	1.79%	1.29%	1.00%	**0.92%**	2.48%
	RTMR	63.78%	33.39%	18.85%	11.55%	**6.12%**
	IMDB	26.17%	2.45%	0.35%	0.04%	**0.00%**
	TOXIC	65.35%	29.34%	17.57%	12.31%	**10.38%**
	MNIST	95.34%	49.20%	26.17%	13.69%	**8.54%**
LSTM	Tomita3	4.92%	0.93%	0.31%	**0.12%**	0.18%
	Tomita4	12.88%	0.86%	0.30%	0.14%	**0.09%**
	Tomita7	12.89%	0.70%	0.15%	0.40%	**0.09%**
	BP	2.18%	0.51%	**0.18%**	0.37%	0.65%
	RTMR	69.03%	40.79%	27.20%	20.56%	**16.78%**
	IMDB	27.36%	2.84%	0.51%	**0.02%**	0.04%
	TOXIC	66.91%	29.81%	18.47%	12.73%	**11.16%**
	MNSIT	90.98%	47.63%	21.72%	14.66%	**9.59%**

6.3 How often do ties in ANS scores occur?

Motivation and Approach. A *tie* [196], which is a set of statements that assigns the same suspiciousness scores, is a well-known problem in SBFL for programs. If faulty and non-faulty statements coexist in a tie, developers have to check all statements in the tie in the worst case scenario. In [196], such a tie is defined as a *critical tie*.

PAFL can consider ties because it selects data samples based on an ANS score. To understand

the impact of the ties on PAFL, we investigate the ties in ANS scores. We consider that a set of samples assigned the same ANS score is defined as an *ANS tie*. Moreover, an ANS tie having both successfully and unsuccessfully predicted data samples is defined as a *critical ANS tie*. If a large number of samples are included in critical ANS ties, developers expend additional effort to classify successfully and unsuccessfully predicted data samples.

We investigate how often critical ANS ties occur. Specifically, after calculating an ANS score for each sample in the validation data, we report the percentage of data samples in critical ANS ties. As in the case of RQ1 and RQ2, we try the parameters $k = 2, 4, 6, 8, 10$ and $n = 1, 2, 3, 4, 5$, and the five SBFL formulas (i.e., Ample, Tarantula, Ochiai, Ochiai2, and DStar).

Result. Table 3.11 shows the percentage of samples in critical ANS ties for each dataset, RNN variant, and k. Because we observed that the n and SBFL formulas do not have as much effect as k on the percentage, the n and SBFL formulas are averaged for each dataset, RNN variant, and k.

As k increases, the percentage of data samples in critical ANS ties decreases. Indeed, the lowest percentages are found when $k = 10$, which is the maximum k value in this study, in 18 out of 24 rows in Table 3.11. k is a parameter related to the number of states of extracted PFAs. The larger the number of states of the PFA, the more diverse the patterns of the n-gram of the states. Hence, we suppose that ties are less likely to occur. Therefore, the problem of critical ANS ties can be alleviated by increasing k. Note, however, that increasing k increases the time needed to extract PFAs from RNNs.

Summary of Discussion3: Ties in ANS scores are observed in various settings, but they can be alleviated by increasing the parameter k, the number of clusters of the PFA construction.

6.4 Are PFAs superior to DFAs in our sample-based fault localization (PAFL)?

Motivation and Approach. PAFL uses PFAs to approximate RNNs by static models. As described in Section 3.2, Dong et al. [40] reported that PFAs are superior to DFAs as models extracted from RNNs. However, this prior study did not compare PFAs and DFAs in terms of sample-based fault localization.

This section clarifies whether using PFAs instead of DFAs enhances the performance of our proposed sample-based fault localization, PAFL. Specifically, we compare two PAFL variants that use either PFAs or DFAs in terms of the prediction performance on the selected data samples. As a method for extracting DFAs from RNNs, we use the method of Hou et al. [66], which has been compared to a baseline in the study of PFA extraction [40]. We show the results obtained using a parameter setting (i.e., k, n, and SBFL formulas) for which RNNs exhibit the worst performance

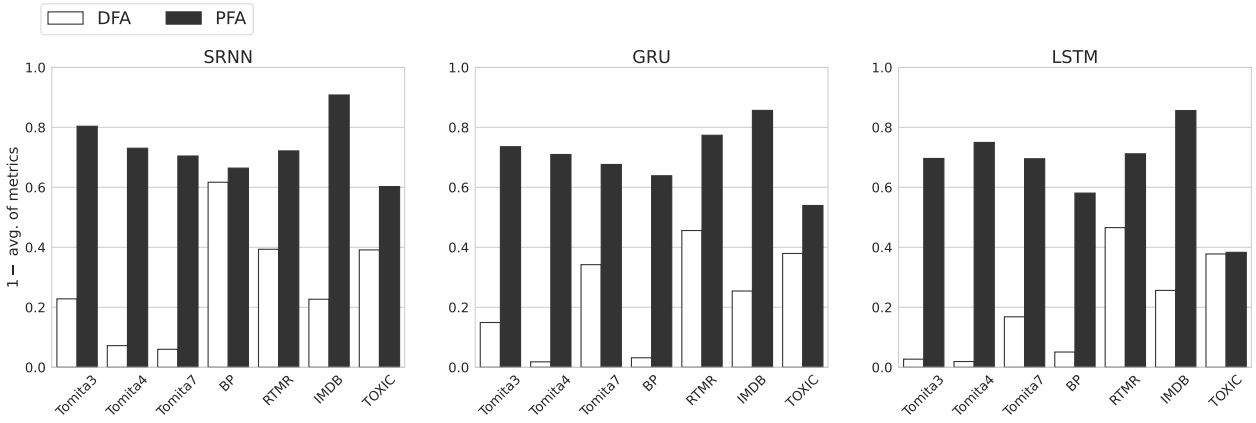


Figure 3.10: Difficulty in predicting the data samples selected by PAFL with PFAs and PAFL with DFAs.

on the selected data samples by PAFL for each dataset and RNN variant.

Result. Figure 3.10 shows how difficult it is to predict the selected samples for each dataset and RNN variant when using PFAs and DFAs. We show the results obtained with a parameter setting (i.e., k, n, and SBFL formula) for which PAFL selects data samples that result in the worst prediction performance for each dataset and RNN variant. The x-axis shows the dataset, and the y-axis shows the values obtained after subtracting the average of the six evaluation criteria for the selected samples from one, which represent the difficulty of predicting the selected samples. The higher the value on the y-axis, the better PAFL more effectively selects samples that cause misbehavior of RNNs.

For all datasets and RNN variants, PAFL with PFAs outperforms one with DFAs. In most settings, PAFL with PFAs has a y-axis value of 0.6 or greater, while PAFL with DFAs has a y-axis value of 0.4 or less. These results indicate that PFAs are superior to DFAs as models extracted from RNNs in sample-based fault localization. This is due to the difference in the ability to represent RNNs behavior faithfully. Indeed, we confirmed that PFAs showed higher fidelity (i.e., the percentage of the same predicted label between RNNs and extracted automata) than DFAs, as confirmed in a previous study [40].

Summary of Discussion4: PFAs are superior to DFAs in our sample-based fault localization, PAFL. For all datasets and RNN variants, PAFL with PFAs can select data samples in which RNNs result in worse prediction performance than PAFL with DFAs.

7 Threats to Validity

7.1 Internal Validity

PAFL extracts PFAs from RNNs as the first step, and PFAs are pivotal for localizing faults; hence, the performance of PAFL depends significantly on how accurately the extracted PFAs reflect the behavior of RNNs. If the extracted PFAs do not reflect the behavior of RNNs, the subsequent steps of PAFL may be meaningless. The method used in this study shows a high fidelity of more than 90% (i.e., the outputs of RNNs and PFAs are the same for more than 90% of the total data samples) [40]; however, PFAs are only an approximation of RNNs. We can justify the reason for using this method and PFAs from two viewpoints. First, converting dynamic models (RNNs) into static models (PFAs) is one key direction for handling complex deep learning architectures. Indeed, prior studies [194, 40] use this conversion to handle RNNs. While this is not the perfect approach, it provides a reasonable approximation. Second, our experiments empirically show that PAFL can effectively localize faulty samples on all the investigated RNN variants and outperforms the state-of-the-art fault localization approach [194], which also uses a similar conversion. However, future studies are necessary to improve the approach to localizing faults on PFAs and the method of extracting approximations from RNNs.

In this study, we used SBFL as the fault localization technique. However, various fault localization techniques exist, such as *mutation-based*, *slicing-based*, and *learning-based fault localization techniques* [186]. Future studies are necessary to clarify the best fault localization technique to improve the performance of PAFL. We suppose we can apply such techniques to PAFL. For example, if we define a mutation operation on PFAs (e.g., changing the probability of a state transition), we can apply *mutation-based fault localization* to PAFL. Defining the effective mutation operations and measuring to what extent such operations affect the performance of PAFL would be worth exploring.

Bootstrap sampling was conducted to statistically evaluate the effectiveness of PAFL. The number of bootstrap samples was ten in our experiment owing to the limitation of the training time of RNNs. Hence, increasing the number of bootstrap samples would alleviate data selection bias, and the results of this study would be more robust.

We provided all scripts used in our study as a replication package (Section 1). This package allows anyone to replicate and validate the results.

7.2 External Validity

We investigated the effectiveness of PAFL using eight datasets consisting of artificial/natural languages and images. However, we avoided using datasets, such as Tomita1, 2, 5, and 6. These datasets were not available because the method used to extract PFAs from RNNs [40] fails to extract states that represent the minority label in imbalanced datasets. Further studies are necessary to investigate the impact of PAFL on such imbalanced datasets.

8 Conclusion

It is challenging to localize and correct faults in RNNs, which have dynamically changing states. In this paper, we proposed a novel fault-localization technique called *probabilistic automaton-based fault localization (PAFL)* for RNNs. PAFL transforms RNNs into PFAs and localizes the sequences of state transitions on PFA as faults of RNNs with n-grams. Considering n-grams, PAFL can handle various internal states of RNNs. Additionally, we distinguished data samples that contribute to faults in evaluating the PAFL. We proposed a novel suspiciousness score called *average n-gram suspiciousness (ANS) score*, based on n-grams and PAFL to distinguish data samples that contribute to faults.

We evaluated PAFL and ANS scores on eight public datasets. The main findings of this study are as follows:

- PAFL distinguishes data samples that cause the misbehavior of RNNs. Specifically, PAFL distinguishes such data samples more accurately on datasets for which RNNs present poor performance. Moreover, PAFL outperforms a state-of-the-art fault localization technique.
- ANS scores represent the misbehavior of RNNs. Additionally, considering state transitions with an n-gram is important for computing ANS scores.

Chapter 4

Repairs and Breaks Prediction for Deep Neural Networks

1 Introduction

Recently, software systems incorporating deep neural networks (DNNs) have become widespread. These systems are used in various applications such as autonomous driving [28, 202, 47], medical image analysis [106, 64, 170], and programming [16, 30]. Misbehavior of DNNs in these systems leads to serious accidents and losses, as seen in accidents in autonomous driving systems [4, 3]. Therefore, repairing the misbehavior of DNNs has become a crucial issue in both the software engineering (SE) and machine learning (ML) communities.

Various methods have been proposed to repair DNNs so far. These methods can be categorized into two types: *model-level repair* that targets components of the DNN model itself (e.g., neurons or weights) [111, 45, 205, 200, 171, 165, 177, 149, 23, 104, 72, 174], and *source code-level repair* that targets the source code used to train the DNN [183, 212, 182, 24, 89, 53]. Arachne [165] is a representative method in the former category, which identifies weights related to misbehavior and uses evolutionary computation to generate repaired values for those weights. *AutoTrainer* [212], on the other hand, is a representative method in the latter category, which monitors the gradient and activation function values during training and changes the source code to repair. The scope of our study is model-level repair for two reasons. First, it has been reported that source code-level repair is often not as effective as ground truth fixes [89]. Second, model-level repair is a promising direction expected to be adopted by the industrial field. For example, a case study [128] investigates the feasibility of model-level repair methods in the industry. This study has identified the requirements and challenges of DNN updating for industrial application (e.g., as few regressions

as possible).

Existing repair methods do not always provide the desired results. The "desired" results vary depending on the developer applying the repair method. In certain cases, the highest priority may be repairing misbehavior, while in others, it may be maintaining correct behavior. When the developer's desired results are not achieved, the execution times, computational resources, and economic costs spent on running the repair method are wasted. Unfortunately, existing methods sometimes suffer from long execution times and poor repair performance as we found in our preliminary experiments (Section 2).

In this study, we build models that predict the repair outcomes to help determine in advance whether the repair method is likely to work. Specifically, we build two types of prediction models: (1) predicting whether a misclassification[1] can be corrected by a repair method, and (2) predicting whether a correct classification can be changed to misclassification by a repair method. We refer to these prediction models as *repairs prediction models* and *breaks prediction models*, respectively. These prediction models use the confidence and ambiguity of the pre-repaired DNN outputs for each sample as explanatory variables. The training dataset for these models is the past repair history, i.e., whether a misclassified sample is changed to the correct classification after repair or whether a correctly classified sample is changed to a misclassification after repair. We use three types of model implementations to enable fast training and inference: logistic regression, random forest, and LightGBM.

Our proposed method can boost the DNN repairing in practice as shown in Figure 4.1. Repairs and breaks prediction models can be built using the limited data samples available during development. These prediction models allow developers to estimate the number of misclassified samples that are likely to be corrected and the number of correctly classified samples that are likely to be broken without applying a time-consuming repair method. In practical use, developers can use these prediction models to predict repairs and breaks for data samples faced during operation, and choose the repair method that best meets their needs. Although the number of data faced during operation is unlimited, our prediction models work with little overhead due to their fast implementation. We demonstrate that these prediction models are useful for developers and operators of DNNs as they assist in deciding whether or not to apply a repair method and, if so, which method to use.

We conducted experiments on various targets, including 10 datasets, three DNN architectures, and four representative repair methods to analyze the repair results and investigate the performance of the repairs and breaks prediction models. The 10 datasets are divided into three types:

[1]We focus on DNNs for the classification task in this study since most of the existing repair methods do so.

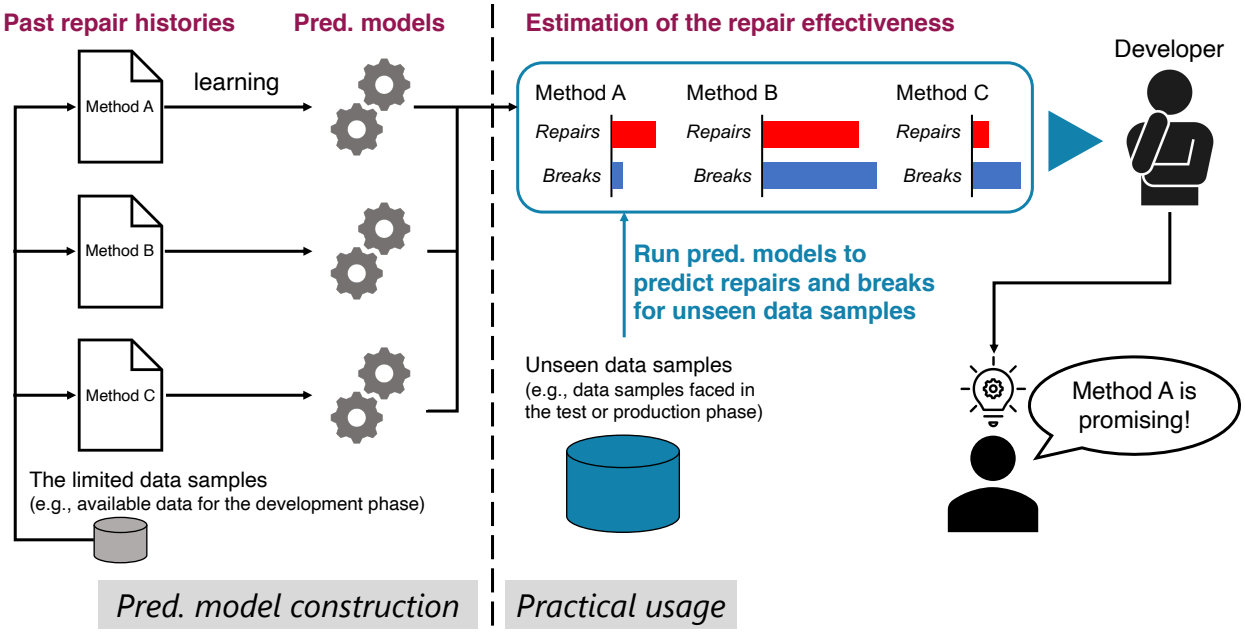


Figure 4.1: The flow for construction and practical usage of repairs and breaks prediction models.

tabular, image, and text datasets. The three architectures are feedforward neural networks (FNNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). As for the target repair methods, we select four representative methods: CARE [171], Apricot [205], Arachne [165], and APRNN [174]. These methods have been published in top-tier international conferences and journals in the SE community.

The contributions of this study are as follows:

- We apply existing repair methods to various targets, including 10 datasets, three DNN architectures, and four repair methods. Statistical analysis of the repair results reveals statistically significant differences in the values representing the confidence and ambiguity of the DNN's predictions between the misclassified (correctly classified) samples that become correctly classified (misclassified) ones and those that do not.

- We build lightweight prediction models to predict repair outcomes using metrics that represent the confidence and ambiguity of DNN's predictions. These models require minimal training and inference time. Furthermore, our investigation demonstrates that these models can also predict repair outcomes for unknown repair methods.

- Our prediction models can suggest suitable repair methods to developers based on their needs (e.g., represented by the ratio of samples that are repaired) with little overhead. Our method identifies the most promising repair method in 16 out of 24 cases, resulting in an

average time savings of 16.29% compared to the naive method of applying all repair methods and checking the results.

- In addition to the most basic property of correctness, we evaluate the generalizability of our method to multiple properties of DNNs (i.e., robustness, fairness, and safety).
- We provide a replication package and curate datasets of repair history for future use by researchers and practitioners.[2]

This paper is an extended version of our previous work titled "*An Initial Analysis of Repair and Side-effect Prediction for Neural Networks*" [73], which was published as a short paper at the 2023 IEEE/ACM 2nd International Conference on AI Engineering - Software Engineering for AI (CAIN). This extended version includes the following distinct differences:

- **Expansion of experimental subjects.** In the previous version, we only focused on CNNs as the target architecture and Arachne as the target repair method. In this extended version, we have expanded the scope to include different architectures, repair methods, and datasets to conduct more comprehensive experiments.
- **Expansion of research questions.** The previous version only built prediction models and evaluated their prediction performance (corresponding to RQ2 in this paper). In this version, we have enhanced the support for the method through statistical analysis of the metrics used in prediction (RQ1). Additionally, we have demonstrated the time-saving benefits of our method for developers and operators of DNNs (RQ3). Moreover, we have investigated the transferability of the prediction model to unknown repair methods, demonstrating the generalizability of our method to future repair methods (RQ4). Furthermore, we have conducted an experiment to explore the applicability of our method in terms of other DNN properties (i.e., robustness, fairness, and safety) (RQ5).

The rest of this paper is organized as follows. Section 2 describes the motivation for our study. Section 3 presents the method for predicting the repair results. Section 4 explains the experimental subjects and Section 5 describes the experimental setup such as the research questions and experimental flow. Section 6 shows the experimental results to answer the research questions. We discuss the results of our study in Section 7. Section 8 explains the threats to validity of our study. Section 9 summarizes our study.

[2] https://github.com/posl/DNN-repair-break-prediction

2 Motivation

To motivate our study, we conducted preliminary experiments applying three representative repair methods (i.e., CARE, Apricot, and Arachne) to neural network models trained on eight datasets consisting of tables, images, and texts. For each dataset, we prepared one neural network model. We evaluated the performance of three repair methods, as well as their execution time.

The results are presented in the Table 4.1. These results are averaged over multiple repair runs and detailed experimental settings are described in Section 5. As indicated in the footnotes of Table 4.1, this experiment was also conducted on datasets that were not used in the previous papers [171, 205, 165]: IMDB and RTMR. In the table, "%repaired" indicates the percentage of existing misclassifications that were corrected by a repair method, while "%broken" indicates the percentage of existing correct classifications that changed to misclassification by a repair method. The detailed definitions of repairs and breaks are explained in Section 3.

CARE has a very low %repaired and %broken, less than 10%, indicating that it is a conservative repair method. Apricot has a higher %repaired than the other two, but a very high %broken on the C10 dataset, indicating that it is unstable. Apricot also has a longer execution time due to the training involved. The time cost is particularly high for the image datasets (i.e., FM, C10, GTSRB), which took more than five hours. Arachne has a good balance between %repaired, %broken, and execution time, but tends to have a higher %broken on tabular datasets.

From these results, our findings are as follows:

- Different repair methods have different characteristics of repair results.
- Even when the %repaired is high, the %broken may be high.
- The method with an additional training process (i.e., Apricot) takes longer than the other methods.

The results of repairs and breaks are not known until the repair method is applied. This is especially challenging for time-consuming methods like Apricot. Even for the other two methods, estimating repairs and breaks in a short time can help in selecting the most likely method to achieve desired results. Hence, the objective of this study is to address these concerns by predicting the performance of repair method in advance.

Summary of our motivation: Preliminary experimental results show that existing repair methods are not perfect. To predict repairs and breaks in advance, we can utilize the characteristics of the repair outcomes for each repair method. The ability to predict these outcomes

Table 4.1: Performance and execution time of the three repair methods.

		CARE			Apricot			Arachne		
Type	Dataset	%repaired	%broken	exe. time	%repaired	%broken	exe. time	%repaired	%broken	Exe. time
Tabular	Credit♦	0.68%	0.27%	0m16s	62.90%	11.60%	0m51s	40.11%	28.66%	0m09s
	Census♦	1.52%	0.48%	2m59s	68.41%	7.80%	2h23m03s	59.57%	37.63%	2m04s
	Bank♦	1.31%	0.31%	4m04s	71.97%	4.30%	3h24m57s	59.14%	21.29%	3m12s
Image	FM♦,♣	3.44%	0.07%	1h04m42s	54.02%	1.80%	8h13m53s	20.82%	1.44%	48m15s
	C10♢,♣	7.16%	0.83%	28m03s	60.72%	96.28%	7h10m39s	10.89%	5.43%	1h32m25s
	GTSRB♦,♣	8.86%	0.12%	10m52s	68.36%	1.08%	5h16m07s	2.58%	0.05%	47m48s
Text	IMDB	0.27%	0.00%	0m18s	25.41%	1.07%	7m27s	47.02%	16.72%	5m43s
	RTMR	0.20%	0.01%	0m45s	25.40%	1.93%	30m43s	50.33%	18.64%	14m36s

♦, ♢, and ♣ denote the dataset used in the CARE paper [171], Apricot paper [205], and Arachne paper [165], respectively.

in advance would assist in deciding whether to apply a repair method and selecting a promising method.

3 Methods

In this section, we provide definitions for repairs and breaks in the neural network considered in this research. We then describe the methods to predict them. Figure 4.2 illustrates the workflow for obtaining the prediction models for repairs and breaks. Our method takes a dataset (a set of data samples) D, a model M to be repaired, and a repair method as inputs. The output is a prediction model of the repairs and breaks of the repair method. This section takes the correctness property as an example. However, our method can be extended to other properties of DNNs (e.g., robustness, fairness, and safety), which is investigated in RQ5. Section 4.2 details how to extend concepts of repairs and breaks to these other properties.

3.1 Repairs and Breaks in Neural Network

Repairs and *breaks* are concepts that represent the outcomes of repair methods applied to a neural network M. Repairs are the *intended effect* of the repair method, and breaks are the *side effect* of the repair method. In this section, let $D_{cor} \subseteq D$ be the set of samples correctly classified by M, and $D_{mis} \subseteq D$ be the set of samples misclassified by M. Additionally, let M' be the neural network obtained after applying a repair method. In this paper, M is sometimes referred to as the *original model* and M' as the *repaired model*.

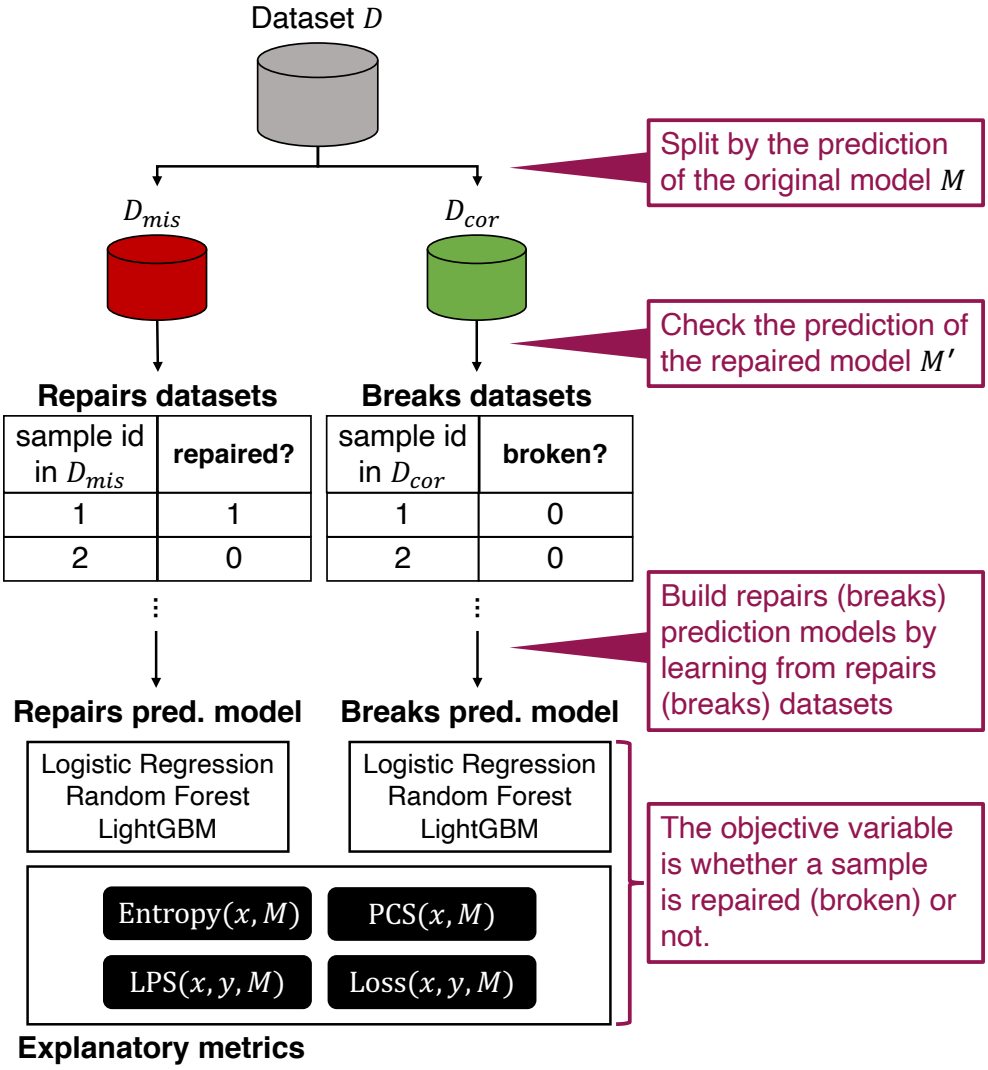


Figure 4.2: Workflow to build repairs and breaks prediction models.

Repairs indicate whether D_{mis} is correctly classified after a repair method is applied. On the other hand, *breaks* indicate whether D_{cor} is misclassified after a repair method is applied. We observe these phenomena through the samples. Among the samples in D_{mis}, the ones that M' can correctly classify are referred to as *repaired samples*, while the samples that are misclassified are called *non-repaired samples*. Similarly, among the samples in D_{cor}, those that are misclassified by M' are referred to as *broken samples*, and the ones that are correctly classified are referred to as *non-broken samples*. We show this concept in Table 4.2. As shown in this table, there are four patterns as to whether each sample is correctly classified before and after repair. Therefore, there are four different outcomes (i.e., non-repaired, repaired, broken, and non-broken samples).

Table 4.2: The definition of repairs and breaks through samples.

Original model M	Repaired model M'	Outcome
incorrect	incorrect	Non-repaired
incorrect	correct	Repaired
correct	incorrect	Broken
correct	correct	Non-broken

3.2 Repairs and Breaks Datasets

We construct the *repairs dataset* and *breaks dataset* that contain the repair application history (tables in the middle of Figure 4.2). These datasets are used to predict repairs and breaks. Here, we explain the repairs and breaks datasets.

The repairs dataset indicates whether each sample in D_{mis} is a repaired sample (labeled as 1) or a non-repaired sample (labeled as 0). On the other hand, the breaks dataset indicates whether each sample in D_{cor} is a broken sample (labeled as 1) or a non-broken sample (labeled as 0). We obtain these repairs and breaks datasets for each repair method. The purpose is to account for the variations in trends of repairs and breaks across different repair methods.

3.3 Repairs and Breaks Prediction Models

In this section, we provide a detailed explanation of the *repairs prediction model* and *breaks prediction model*. Since we cannot observe repaired and broken samples without applying a repair method, our prediction models aim to predict these samples in advance.

Both the repairs and breaks prediction models are binary classification models. The repairs prediction model predicts whether each sample in D_{mis} will be a repaired sample or not after applying a repair method. On the other hand, the breaks prediction model predicts whether each sample in D_{cor} will be a broken sample or not after applying a repair method. The requirements for these prediction models are as follows:

1. Use only explanatory metrics that can be obtained before applying a repair method.
2. Use modeling techniques that allow for fast learning and inference.

Below we explain how these requirements are met.

Explanatory Metrics

We describe the four explanatory metrics for the repairs and breaks prediction models. We use the following four metrics that represent confidence or ambiguity in the prediction of M for each sample because we assume that they affect repairs and breaks. In fact, the previous studies on test input prioritization for DNNs [185] and the classification of benign and adversarial samples [211] have been successful in capturing the characteristics of the DNN predictions for each sample with confidence and ambiguity metrics. Although the objective of this study is different from those, we consider we can refer to these precedents. Since these metrics can be computed by the original model M, the first requirement mentioned above is satisfied. Let x denote each sample in a dataset (i.e., D_{mis} or D_{cor}) and y denote the ground truth for x.

Entropy. Entropy indicates the level of ambiguity in a prediction. We use the well-known Shannon entropy [161].

$$\text{Entropy}(x, M) = -\sum_{i \in C} (P_x[i] \log P_x[i]) \tag{4.1}$$

where $P_x[i]$ represents the prediction probability that the model M predicts that x belongs to label i. C is the set of labels.

Prediction Confidence Score (PCS). PCS is defined as the difference between the highest prediction probability and the second-highest prediction probability [211]. A value close to one indicates high confidence in the model's prediction, while a value closer to zero indicates low confidence. The PCS for a sample x of a model M is calculated as follows:

$$\text{PCS}(x, M) = P_x[c_{pred}] - \max_{i \in C \setminus c_{pred}} P_x[i] \tag{4.2}$$

where c_{pred} represents the label predicted by M, which is the label with the highest prediction probability.

Label Prediction Score (LPS). We named LPS the prediction probability for the ground truth [73].

$$\text{LPS}(x, y, M) = P_x[y] \tag{4.3}$$

Loss. Loss refers to the value of the loss function during prediction. In this study, we use binary cross entropy [22] as the loss function for binary classification models, and categorical cross entropy [126] for multiclass classification models.

$$\text{Loss}(x, y, M) = \begin{cases} \text{BinaryCrossEntropy}(x, y, M) & (|C| = 2) \\ \text{CategoricalCrossEntropy}(x, y, M) & (|C| \geq 3) \end{cases} \tag{4.4}$$

Importantly, these metrics can be applied to any model that outputs probabilities for each class (i.e., any classification model). In other words, they are general metrics that are independent of the type of dataset, such as tables, images, text, etc. This makes it possible to apply our method to a variety of tasks.

Model Implementation

Here we describe how to construct repairs and breaks prediction models. Since our goal is to quickly predict repairs and breaks before applying a repair method (as stated in the second requirement), we use three types of implementations of the prediction models: *Logistic Regression (LR)*, *Random Forest (RF)*, and *LightGBM (LGB)*. To build the repairs (breaks) prediction models, 80% of the total repairs (breaks) datasets are used as training and validation sets. The remaining 20% is used as a test set to evaluate the performance of the models. For training and validation sets, we perform five-fold cross-validations to tune the hyperparameters of these models. The evaluation of the models is based on six metrics: accuracy (acc.), precision (pre.), recall (rec.), f1-score (f1.), Receiver Operating Characteristic Area Under the Curve (ROC-AUC), and Precision-Recall Area Under the Curve (PR-AUC). This is because the tasks for repairs and breaks prediction models are binary classifications.

4 Experimental Subjects

4.1 Datasets

We use 10 datasets consisting of three types of datasets (i.e., tabular, image, and text datasets) to evaluate the effectiveness of our method. Two of these datasets are for robustness evaluation, which are used only for RQ5. We believe that these datasets encompass the basic types of datasets for which neural networks are commonly employed. Most of the existing studies focus mainly on image datasets. At most, their experiments have been conducted on two or three types of datasets (e.g., image and text datasets). Therefore, for the sake of a comprehensive study, we subject the three types of datasets.

Tabular Datasets

- **German Credit (Credit) [65]**. This dataset is used to classify good and bad credit risks. It consists of a total of 1,000 instances in total and 20 features. The labels are binary values that indicate whether the individual credit risk is good (1) or bad (0).

- **Census Income (Census) [93].** This dataset is used to predict whether income exceeds $50K per year. It consists of a total of 32,561 instances and 14 features. The labels are binary values that indicate whether the individual income exceeds $50K (1) or not (0).
- **Bank Marketing (Bank) [124].** This dataset is used to predict whether the client subscribes a term deposit. It consists of a total of 45,211 instances and 16 features. The labels are binary values that indicate whether the client subscribes a term deposit (1) or not (0).

Image Datasets

- **Fashion-MNIST (FM) [193].** This dataset is for the classification of fashion images. It has 60,000 training samples and 10,000 test samples. Each sample is a 28x28 grayscale pixel value. The labels are numbers from zero to nine that indicate a category of fashion items.
- **CIFAR-10 (C10) [94].** This dataset is for a 10-class classification. It has 50,000 training samples and 10,000 test samples. Each sample is 32x32 in size and has three channels (i.e., RGB). The labels are numbers from zero to nine indicating each category, such as bird, cat, and automobile.
- **German Traffic Sign Recognition Benchmark (GTSRB) [168].** This dataset is for the classification of traffic signs. It has 39,209 training samples and 12,630 test samples. Each sample has three channels and is resized to 48x48 as in the prior study [165]. There are 43 labels indicating traffic signs, such as speed limit and stop signs.

Text Datasets

- **IMDB [112].** This is a natural language dataset for sentiment analysis of movie reviews. The dataset consists of review sentences, and each review sentence is labeled as positive (1) or negative (0). We extracted 3,730 training samples and 933 test samples according to the preprocessing performed in existing studies [72].
- **RTMR [142].** This dataset, as well as IMDB, is a dataset for sentiment analysis of movie reviews. The dataset has 8,530 training samples and 2,130 test samples.

Robustness Datasets

- **Fashion-MNIST-C (FMC) [185].** The FM dataset with various corruptions such as noise, blur, transformations, etc. The dataset size is the same as the original Fashion-MNIST.

- **CIFAR-10-C (C10C) [63].** A total of 19 corruptions (noise, blur, weather, etc.) are applied to the C10 dataset, with 50,000 samples for each of the 19 corruptions. Since the total dataset size is too large, we treat the 50,000 samples for each corruption as a single dataset.

We chose these two datasets for two main reasons. First, these datasets are specifically designed to simulate real-world corruptions in image data, such as noise, blur, and weather. This makes them suitable for evaluating the robustness of models. While an alternative approach is to evaluate robustness using adversarial attacks [54, 15], we opted not to use adversarial samples because they may represent "unnatural" samples that would not occur in the real world, and they can be costly to generate. Second, these datasets are compatible with the datasets used in our experiments (i.e., FM and C10). FMC and C10C are provided for robustness evaluation of models trained with FM and C10, respectively. Specifically, the classification performance of models on FMC and C10C datasets serves as a measure of the robustness of models trained on FM and C10, respectively. That is, we assess robustness by examining how well models trained on an uncorrupted dataset can predict a corrupted dataset. The point is that we can evaluate robustness in the same way as correctness, which means that the experiments can be easily extended to the robustness property.

4.2 DNN Properties

We target four types of properties of DNNs. RQ1 through RQ4 target *correctness*, which is the most basic property of DNNs for classification problems, while RQ5 covers other important properties such as *robustness*, *fairness*, and *safety*. This section describes how to extend concepts of repairs and breaks to properties other than correctness.

In preparation for such an extension, we define a generic *property* for a DNN as a function that outputs a binary value for each data sample. Specifically, given a dataset D and a DNN M, the *property function* f_M maps each pair $(x, y) \in D$ to a binary value (i.e., 0 or 1) using M. This can be formally written as follows:

$$f_M : D \to \{0, 1\}\,. \tag{4.5}$$

It indicates whether a DNN M behaves as expected (1) or not (0) for the sample $(x, y) \in D$ under certain properties. This definition is useful for extending our definition of repairs and breaks to properties other than correctness. That is, we can instantiate the above property function f_M by specific properties, i.e., Correctness_M, Robustness_M, Fairness_M, and Safety_M in our experiment.

Correctness

This is the most basic property that indicates whether a DNN correctly classifies a data sample. The property function of correctness can be defined as follows:

$$\text{Correctness}_M(x, y) = \begin{cases} 1 & (M(x) = y) \\ 0 & (M(x) \neq y) \end{cases} . \tag{4.6}$$

We can evaluate this property for all the datasets described in Section 4.1 because they have ground truth labels. However, FMC and C10C are only for robustness evaluation, so we evaluate correctness for the other eight datasets.

Robustness

This property indicates whether a DNN classifies a data sample correctly when corruption is added to the sample. We assume that D^c is the corrupted version of the dataset D. For instance, if D is the FM dataset, then D^c is the FMC dataset. The property function of robustness is the correctness for a corrupted sample $(x^c, y^c) \in D^c$:

$$\text{Robustness}_M(x, y) = \text{Correctness}_M(x^c, y^c). \tag{4.7}$$

Note that the DNN M is trained on D rather than D^c. We evaluate the robustness of DNNs trained on FM and C10 with FMC and C10C, respectively.

Fairness

This property indicates whether a DNN do not make favorable predictions based solely on a *sensitive feature* (e.g., gender, age, and race). The property function of fairness can be defined as follows:

$$\text{Fairness}_M(x) = \begin{cases} 1 & (d(x, M) \geq \theta) \\ 0 & (d(x, M) < \theta) \end{cases} . \tag{4.8}$$

Here, $d(x, M)$ is an *independence-based fairness* [207, 171], which is defined as follows:

$$d(x, M) = 1 - \max_{x'} \left| P(M(x) = y) - P(M(x') = y) \right| , \tag{4.9}$$

where x' is a sample that differs only in sensitive features from x, and y is a favorable prediction label. If $\max_{x'} |P(M(x) = y) - P(M(x') = y)|$ is large, it indicates that M could make unfair predictions because the predictions could vary greatly depending only on the sensitive feature. In our experiment, we subtract this value from 1, so that a larger value of $d(x, M)$ indicates more fair. θ is a threshold value above which the model is considered to be fair for the sample.

We use three tabular datasets (i.e., Credit, Census, and Bank) for the fairness evaluation. These datasets are suitable for this purpose because they contain sensitive features such as gender and age. In our experiment, gender is treated as a sensitive feature in Credit and Census, and age is treated as a sensitive feature in Bank. The three datasets are the same as those in CARE's original paper, so the value of θ in Equation 4.8 is the same as that used in their experiment (i.e., $\theta = 0.95$).

Safety

This property indicates whether a DNN output satisfies a safety condition when applied to safety-critical systems. The property function of safety can be defined as follows:

$$\text{Safety}_M(x) = \begin{cases} 1 & (\phi_i(x, M)) \\ 0 & (\neg\phi_i(x, M)) \end{cases}, \tag{4.10}$$

where ϕ_i is a pre-defined safety condition that M must satisfy for the input x.

We evaluate the safety of the FNNs for the Airborne Collision Avoidance System (ACAS Xu) [82]. ACAS Xu contains 45 FNNs that produce horizontal maneuver advisories. The input of each FNN is five numerical variables, such as the distance and angle to the intruder from the own-ship, and the output is the horizontal maneuver advisory, which consists of five different types. Although the training and evaluation datasets are not publicly available, the weights of each FNN are available. Therefore, we can get the output of each FNN and check the safety conditions by randomly sampling the input values of five dimensions.

Katz et al. [86] defined 10 safety conditions (denoted as $\phi_i (1 \leq i \leq 10)$) for the FNNs for the ACAS Xu. These are the output conditions that the FNNs must satisfy in terms of safety for a given input condition. Previous studies [86, 171] have demonstrated that some FNNs violate some conditions. We replicate their results and confirm that $N_{2,9}$ and $N_{3,5}$[3] violate ϕ_8 and ϕ_2, respectively, for certain inputs. Hence, we use these two FNN and safety condition pairs for our safety evaluation.

Definition of Repairs and Breaks for Generic Properties

As shown in Table 4.3, we can determine repaired, non-repaired, broken, and non-broken for each data sample by comparing $f_M(x, y)$ and $f_{M'}(x, y)$. Here, M is the original model, and M' is the repaired model. Repairs or breaks criteria (i.e., whether the data sample was processed as expected by the DNN) depend on the properties defined in f_M. Note that Section 3 only gives an example of $f_M = \text{Correctness}_M$.

[3] $N_{i,j}$ shows one of the 45 FNNs for ACAS Xu.

Table 4.3: The definition of repairs and breaks through a property function.

$f_M(x, y)$	$f_{M'}(x, y)$	Outcome
0	0	Non-repaired
0	1	Repaired
1	0	Broken
1	1	Non-broken

Table 4.4: Neural network models for each task.

Task	Architecture	#Layers	#Params.	Property*
Credit	FNN	6	4,326	C, F
Census	FNN	6	5,286	C, F
Bank	FNN	6	4,996	C, F
FM	CNN	7	3,274,634	C, R
C10	CNN	9	2,425,930	C, R
GTSRB	CNN	12	1,390,943	C
IMDB	RNN	2	219,394	C
RTMR	RNN	2	219,394	C
ACAS $N_{2,9}$	FNN	6	13,305	S
ACAS $N_{3,5}$	FNN	6	13,305	S

* C: Correctness, F: Fairness, R: Robustness, S: Safety.

4.3 Neural Network Models

We experiment with various types of neural networks for classification: FNNs, CNNs, and RNNs.[4] FNNs are neural networks that consist mainly of a fully connected layer, while CNNs and RNNs have not only a fully connected layer but also specific layers such as a convolutional layer and a recurrent layer, respectively. FNNs, CNNs, and RNNs are commonly used for processing tabular, image, and text datasets, respectively, so we use these architectures of neural networks. All FNNs for ACAS Xu have the same architecture but different weight parameters for each. Table 4.4 describes the neural network models used in our experiments. The rightmost column of this table shows the DNN properties evaluated for each task.

[4]To be precise, we used Long Short-Term Memory networks (LSTMs). However, we refer to them as RNNs in this paper since LSTMs are a variant of the RNN family.

4.4 Repair Methods

As studied repair methods for neural networks, we use four representative repair methods: CARE, Apricot, Arachne, and APRNN. These methods were selected based on the following criteria: (1) they are model-level repair methods, (2) they represent state-of-the-art techniques published in leading conferences and journals, and (3) they are available as open source.[5] Furthermore, as shown in Table 2.1, the subjects cover a variety of model-level repair methods.

5 Experimental Setup

In this section, we present the research questions (RQs) and experimental procedure for addressing these RQs. For each part of the procedure, we provide comprehensive details on the experimental settings.

5.1 Research Questions

RQ1: (Differences) What are the characteristics of repaired and broken samples?

In this RQ, we analyze the differences between repaired and non-repaired samples, as well as between broken and non-broken samples. These differences are measured by the four metrics described in Section 3.3. By conducting statistical tests on these metrics, we reveal the differences between repaired and non-repaired samples, as well as between broken and non-broken samples.

RQ2: (Prediction performance) To what extent can repairs and breaks be predicted?

In this RQ, we build repairs (breaks) prediction models to distinguish repaired and non-repaired (broken and non-broken) samples by using the same metrics as RQ1. We evaluate the prediction performance on the test set of repairs (breaks) datasets.

RQ3: (Usefulness) Can repairs and breaks prediction models suggest a promising repair method in a shorter time compared to a naive method?

This RQ investigates the usefulness of the repairs and breaks prediction models in selecting a promising repair method. When there are multiple candidate repair methods, it is enormously time-consuming to try them all and select the best one. We demonstrates that by using our repairs

[5]CARE: https://github.com/longph1989/Socrates, Apricot: https://github.com/theyoucheng/AIRepair, Arachne: https://github.com/coinse/arachne, APRNN: https://github.com/95616ARG/APRNN

and breaks prediction models, we can select the most promising repair method more efficiently compared to simply applying repair methods without these models.

RQ4: (Transferability) Are repairs and breaks prediction models transferable to other repair methods?

It is expensive to build repairs and breaks prediction models for new repair methods because we have to run the repair methods again. Therefore, this RQ investigates the transferability of repairs and breaks prediction models. Specifically, we investigate how well repairs and breaks prediction models, once built, can be used to predict repairs and breaks for unknown repair methods.

RQ5: (Generality) Can the promising repair method selection be applied to DNN properties other than correctness?

- *RQ5-1: Robustness.*
- *RQ5-2: Fairness.*
- *RQ5-3: Safety.*

This RQ examines our method for various DNN properties, i.e., *robustness*, *fairness*, and *safety*. In the previous RQs, the only property was correctness, whether the DNN predictions were correct or not. However, in real-world applications of DNNs, it is crucial to consider DNN properties other than correctness. For example, in safety-critical domains, robustness [217, 26, 63] is an important property alongside correctness. Hence, we explore whether our framework, which builds repairs/breaks prediction models and selects promising repair methods, can also be applied to these properties.

5.2 Experimental Procedure

The experimental procedure is divided into three parts: (1) Build neural network models, (2) Apply repair methods, and (3) Build repairs and breaks prediction models. The experimental procedure is shown in Figure 4.3.

(1) Build Neural Network Models

In the first part, we build neural networks to be repaired (i.e., original models).[6] The original dataset is divided into three subsets: *training*, *repair*, and *test* sets. Initially, a test set is taken from

[6]For FNNs for ACAS Xu, this part is skipped because the weights of the trained model are available.

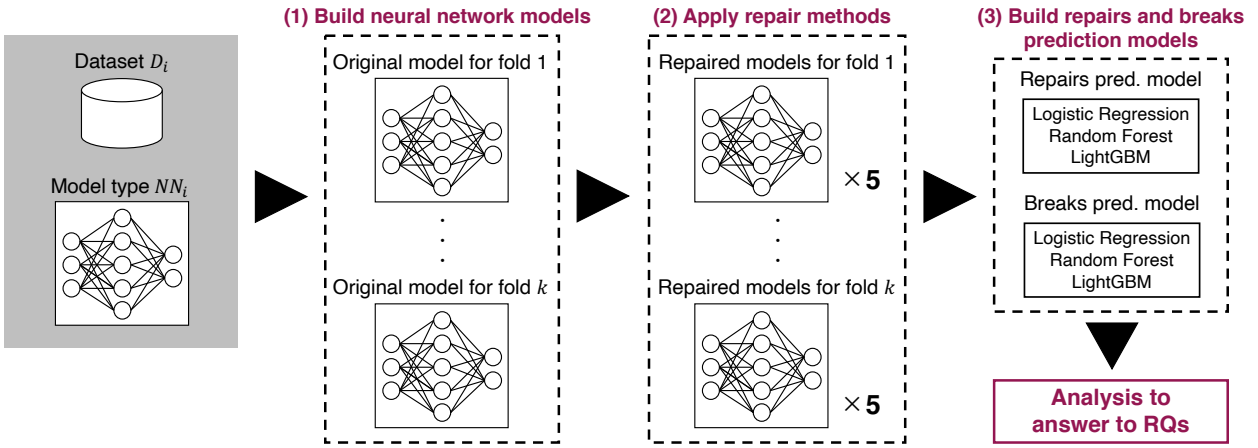


Figure 4.3: The experimental procedure.

the original dataset. The test set can be used only for evaluating the generalization performance of original or repaired models. Subsequently, the portion of the original dataset that does not includes the test set is divided into the training and repair sets. The training set is mainly used to train the neural network, while the repair set is mainly used for repair (some repair methods also use the training set). In this division, as in the cross-validation method, we divide the portion into subsets called folds and iterate to use one fold as a repair set and the rest as a training set. Given that the training set varies depending on how the folds are divided, for each neural network in Table 4.4, we have the same number of original models as the number of folds. This approach serves two purposes: (1) obtaining robust results for the division of training and repair sets, and (2) obtaining a more extensive history for the repair method.

Table 4.5 shows the number of samples and folds for each dataset. For the Credit dataset, the number of folds is 5 and the percentage of test set is 10% to use as many samples as possible for training and repair, since the total number of samples is small. For the other two tabular datasets, the number of folds is 10 and the percentage of test set is 30%, since the number of samples for these datasets is large. For image and text datasets, the number of folds is 5 because training neural networks for these datasets takes longer compared to tabular datasets. In fact, Table 4.4 shows that the number of neural network parameters for text and image datasets is 100 and 1,000 times larger than those for tabular datasets, respectively, resulting in longer training times for text and image datasets. The number of training epochs is 100 for tabular and image datasets and 30 for text datasets. We set 30 epochs for text datasets because the loss decreased sufficiently after 30 epochs of training.

Table 4.5: The number of samples and folds for each dataset.

Dataset	Type	#train	#repair	#test	#folds
Credit	Tabular	720	180	100	5
Census	Tabular	20,512	2,280	9,769	10
Bank	Tabular	28,482	3,165	13,564	10
FM	Image	48,000	12,000	10,000	5
C10	Image	40,000	10,000	10,000	5
GTSRB	Image	31,367	7,842	12,630	5
IMDB	Text	2,984	746	933	5
RTMR	Text	6,823	1,706	2,133	5

(2) Apply Repair Methods

In the second part, the repair methods described in Section 4.4 are applied to the trained neural networks. CARE and Arachne use only the repair set, while Apricot uses the training set as well since this method needs to build reduced models with the training set. This difference between Apricot and others arises because we follow the algorithm proposed in each original paper [171, 205, 165]. All repair methods involve the use of random numbers, leading to different outcomes in each run. To account for this variation, the repair methods are repeated five times for each original model. This means that for each original model built with each fold, we obtain five repaired models. These repaired models are then fed with samples from the training, repair, and test sets, and their outputs are compared to those of the original model. This comparison is used to create repairs and breaks datasets.

(3) Build Repairs and Breaks Prediction Models

In the third part, the repairs and breaks datasets are created based on the results of the repair method obtained in the previous step. The repairs and breaks prediction models are then built as described in Section 3.

To create the repairs and breaks datasets, we use the results of the five repetitions of the repair methods. Specifically, we define the samples in D_{mis} that are correctly classified by at least three repaired models out of five (i.e., the majority of the repaired models) as repaired samples and those that are not as non-repaired samples. We also define the samples in D_{cor} that are misclassified by at least one repaired model out of five as broken samples and those that are not misclassified even once as non-broken samples. These definitions take into account the randomness of the

repair method. The definition of repaired samples reflects whether or not the repair is made in many cases in multiple repair runs. The definition of broken samples, on the other hand, reflects whether there is even a slight chance of breaks.

5.3 Experimental Environment

Our experiments were conducted using Python 3.7, since the target repair methods are implemented in Python. We utilized the PyTorch 1.13.1 for implementing and training the neural network. The hardware used in the experiment is an AWS EC2 instance, which type is p2.xlarge. This instance has a single GPU, Tesla K80.

6 Results

6.1 RQ1: (Differences) What are the characteristics of repaired and broken samples?

Approach. This RQ aims to investigate the differences in four explanatory metrics (i.e., Entropy, PCS, LPS, and Loss) between repaired and non-repaired (broken and non-broken) samples. We analyze the statistical significance and effect size of these metric differences. Specifically, we set the following null hypotheses for repairs and breaks and conduct Mann–Whitney U tests [114].

- H_r: When applying a repair method R to a model trained on dataset D, the difference in metric M between repaired and non-repaired samples is zero.
- H_b: When applying a repair method R to a model trained on dataset D, the difference in a metric M between broken and non-broken samples is zero.

Furthermore, we calculate Cliff's delta d as an effect size from U, the test statistic of the U test [33].

$$d = \frac{2U}{n_1 n_2} - 1 \tag{4.11}$$

where n_1 and n_2 represent the sample sizes of each group. For instance, n_1 can represent the number of repaired samples, and n_2 can represent the number of non-repaired samples. Cliff's delta ranges from -1 to 1, with the absolute value indicating the extent of difference between the two groups, and the sign indicating which group is larger.

Results. Tables 4.6a and 4.6b show the effect sizes and statistical significance of the differences in the four metrics between repaired and non-repaired (broken and non-broken) samples, respectively. The cells in the tables are categorized into large, medium (med.), small, and negligible (neg.)

levels, in descending order of the absolute value of the effect size, based on a previous study [153]. The text color indicates the sign of the effect size: green indicates that the repaired (broken) samples tend to have higher values, while red indicates that the non-repaired (non-broken) samples tend to have higher values.

The repaired samples tend to have statistically significantly higher Entropy and LPS values compared to the non-repaired samples. In Table 4.6a, the Entropy of the repaired samples is statistically significantly higher (indicated by **) in 19 out of 24 cases, and in 20 out of 24 cases for the LPS. However, there is an exception when the dataset is C10 and the repair method is Apricot, where the Entropy of the repaired samples tends to be low. Therefore, this tendency does not hold in all cases. For the image datasets (FM, C10, GTSRB), not only statistically significant differences but also large effect sizes are observed in many cases.

The broken samples tend to have statistically significantly higher Entropy and Loss values compared to the non-broken samples. In Table 4.6b, both the Entropy and Loss of the broken samples are statistically significantly higher (indicated by **) in 21 out of 24 cases. Similar to Table 4.6a, the image datasets show a greater effect size, suggesting distinct differences between broken and non-broken samples for these datasets. Besides, the PCS and LPS values tend to be statistically significantly lower for broken samples. Table 4.6b has more occurrences of "large**" compared to Table 4.6a, indicating a clearer difference between broken and non-broken samples than between repaired and non-repaired samples.

In addition, Figure 4.4 visualizes the differences between repaired and non-repaired (broken and non-broken) samples. We averaged the values of the metric for each label (i.e., repaired, non-repaired, broken, and non-broken) across all datasets. We also performed min-max scaling to keep the values of each metric within the range $[0, 1]$ to draw radar plots. These radar plots are useful for visually understanding the characteristics mentioned above. For instance, the radar plot in Figure 4.4a shows that the repaired samples tend to have higher Entropy and LPS values than the non-repaired samples. However, there are cases where there appears to be no difference between repaired and non-repaired (broken and non-broken) samples, e.g., concerning the radar plot for Apricot shown in Figure 4.4b, the averages of the metrics are almost the same between broken and non-broken samples. This result may be because the values are averaged across all datasets. The trends for each dataset can be seen in Tables 4.6a and 4.6b.

Answer to RQ1: Our statistical investigation reveals that: (1) Misclassified samples with high ambiguity and prediction probability for the correct label (i.e., high Entropy and LPS) are more likely to be repaired. (2) Correctly classified samples with high ambiguity (i.e., high

Table 4.6: Effect sizes and U-test results for the differences in metrics between (a) repaired and non-repaired samples and, (b) broken and non-broken samples.

(a) Repaired vs. non-repaired samples.

	CARE				Apricot				Arachne			
	Entropy	PCS	LPS	Loss	Entropy	PCS	LPS	Loss	Entropy	PCS	LPS	Loss
Credit	med.	med.	med.	med.	med.**	med.**	med.**	med.**	med.**	med.**	med.**	med.**
Census	small**	small**	small**	small**	med.**	med.**	med.**	med.**	neg.	neg.	neg.	neg.
Bank	neg.**	neg.**	neg.**	neg.**	large**	large**	large**	large**	small**	small**	small**	small**
FM	large**	large**	large**	large**	med.**	med.**	large**	large**	large**	large**	large**	large**
C10	large**	large**	large**	large**	small**	small**	large**	large**	med.**	med.**	large**	med.**
GTSRB	large**	large**	large**	large**	neg.*	neg.**	large**	large**	med.**	large**	large**	large**
IMDB	neg.	neg.	large	large	small**	small**	small**	small**	small**	small**	small**	small**
RTMR	large**	large**	neg.	neg.	small**	small**	small**	small**	small**	small**	small**	small**

* denotes significant differences with $p < 0.05$ by the test on H_r, and ** denotes significant differences with $p < 0.01$.
Green indicates that repaired samples tend to have higher values of metrics, and red indicates that non-repaired samples tend to have higher values of metrics.

(b) Broken vs. non-broken samples.

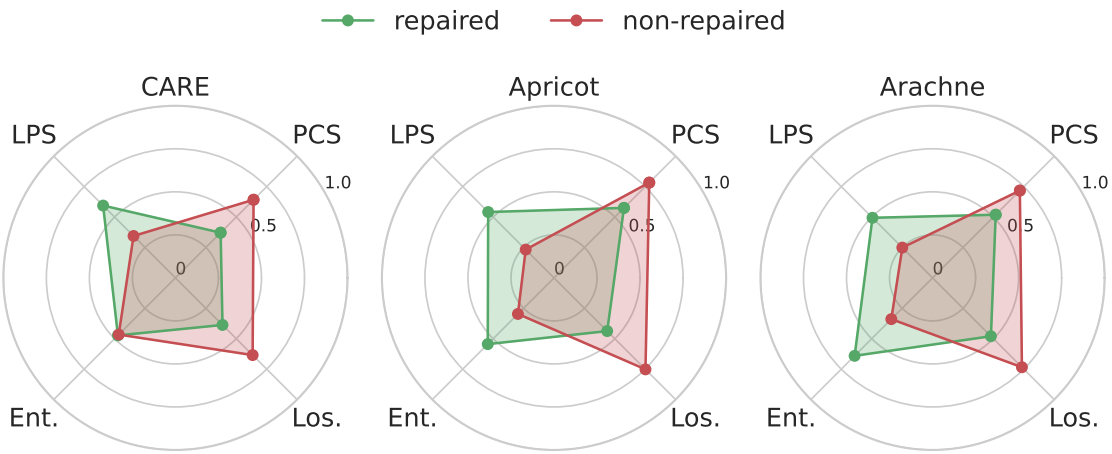
	CARE				Apricot				Arachne			
	Entropy	PCS	LPS	Loss	Entropy	PCS	LPS	Loss	Entropy	PCS	LPS	Loss
Credit	small	small	small	small	med.**	med.**	med.**	med.**	small**	small**	small**	small**
Census	large**	large**	large**	large**	large**	large**	large**	large**	med.**	med.**	med.**	med.**
Bank	large**	large**	large**	large**	large**	large**	large**	large**	large**	large**	large**	large**
FM	large**	large**	large**	large**	large**	large**	large**	large**	large**	large**	large**	large**
C10	large**	large**	large**	large**	small**	neg.**	neg.**	neg.**	large**	med.**	med.**	med.**
GTSRB	large**	large**	large**	large**	large**	large**	large**	large**	large**	large**	large**	large**
IMDB	large	large	large	large	large**	large**	large**	large**	neg.**	neg.**	neg.**	neg.**
RTMR	large	large	large	large	large**	large**	large**	large**	neg.**	neg.**	neg.**	neg.**

* denotes significant differences with $p < 0.05$ by the test on H_b, and ** denotes significant differences with $p < 0.01$.
Green indicates that broken samples tend to have higher values of metrics, and red indicates that non-broken samples tend to have higher values of metrics.

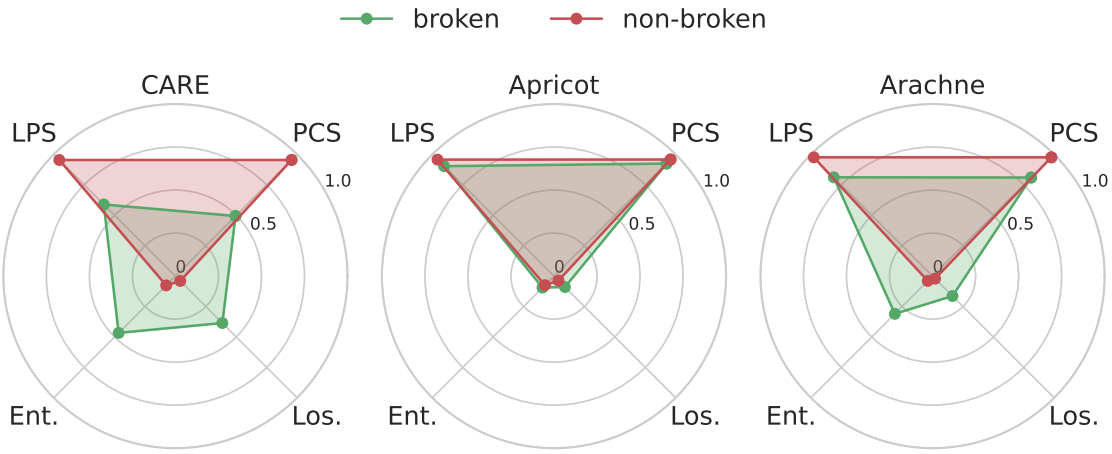
Entropy and Loss) are more likely to be broken.

6.2 RQ2: (Prediction performance) To what extent can repairs and breaks be predicted?

Approach. We use the same metrics as RQ1 as explanatory metrics to train repairs and breaks prediction models and investigate their prediction performance. It is important to note that these prediction models are trained for each dataset and repair method.



(a) Repaired vs. non-repaired samples.



(b) Broken vs. non-broken samples.

Figure 4.4: Comparison of the four metrics between (a) repaired and non-repaired samples, and (b) broken and non-broken samples (correctness property). Each radar plot represents the average of the four metrics across all datasets.

We perform *random under-sampling (RUS)* when training repairs and breaks prediction models because repairs and breaks datasets are often imbalanced. For example, if the repair ratio is 10%, the repairs dataset is imbalanced, as there are nine times as many negative examples (i.e., non-repaired samples) as positive examples (i.e., repaired samples). Thus, RUS is performed to reduce the samples in the majority class. Specifically, if the ratio of the number of samples in the majority class to that in the minority class is five or more, then only 20% of the samples in the majority class are randomly picked. This can mitigate imbalances, but it cannot resolve extreme imbalances, such as when there are only repaired samples of less than 1% (e.g., the CARE cases in Table 4.1). Note that RUS is only applied to the training set and the test set is not modified. The impact of

the resampling strategy is additionally examined in Section 7.3.

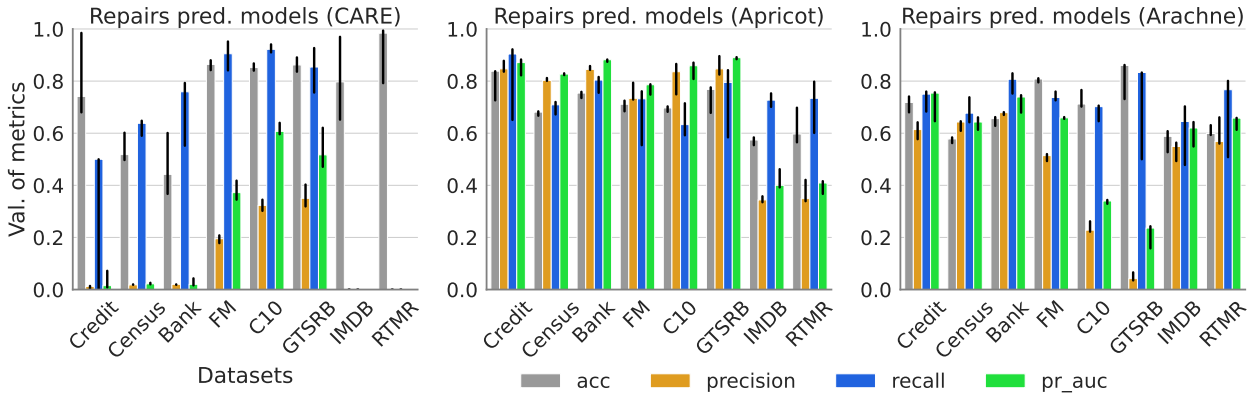
Results. Figures 4.5a and 4.5b show the prediction performance of the repairs and breaks prediction models, respectively. We evaluated their prediction performance using the test set of the repairs or breaks datasets. Since there are three different implementations of the prediction models (i.e., LR, RF, and LGB), the height of each bar indicates the median value of evaluation metrics for each implementation. The error bars represent the range of the minimum and maximum values for the three implementations.

The repairs prediction models for Apricot and Arachne demonstrate a good balance between precision and recall. However, for CARE, the precision is low while the recall is high, indicating an imbalance in the models. Figure 4.5a shows that the repairs prediction models for Apricot achieve a PR-AUC exceeding 0.8 in five datasets, excluding FM, IMDB, and RTMR. This suggests that the predictions are unbiased and show good prediction performance. The models for Arachne show PR-AUC ranging from 0.6 to 0.8. While the evaluation metrics for Arachne are not as high as those for Apricot, they still demonstrate comparable prediction performance. In contrast, the models for CARE exhibit very low precision, with values below 0.4 for all datasets. However, the accuracy and recall values remain high, suggesting a bias in the prediction results.

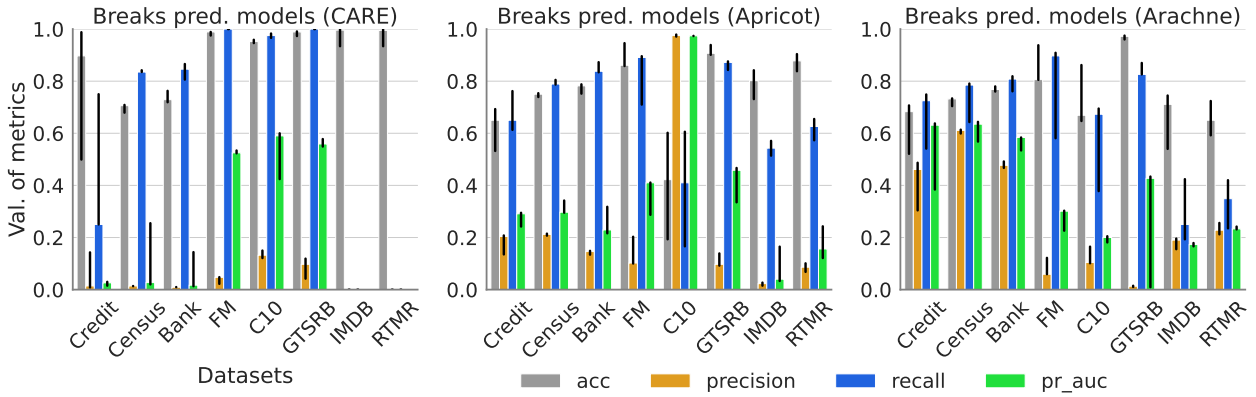
One possible cause for this issue is the imbalance in the datasets used to train the repairs prediction models. Preliminary experiments revealed that CARE had a lower proportion of repaired samples compared to other methods (cf. Table 4.1). This lack of learning from limited positive examples likely resulted in the observed low precision. As an extreme example, the test sets of the repairs datasets for the IMDB and RTMR did not include any repaired samples. Since the test set only had one label, both recall and precision were inevitably 0, making it impossible to calculate ROC-AUC and PR-AUC. On the other hand, the other two methods did not have such a severe imbalance, leading to consistent evaluation metrics across the board. This imbalance is believed to be the underlying cause of the observed differences.

Overall, the breaks prediction models sacrifice precision to achieve high recall. Figure 4.5b shows that, for most repair methods and datasets, the recall falls within the range of 0.6 to 1.0, while the precision remains below 0.5. This trend is especially noticeable for CARE, where the precision consistently stays below 0.2 across all datasets. Our preliminary experiments indicated that broken samples were less common than repaired samples. Consequently, the breaks dataset becomes imbalanced for all repair methods, leading to these results. However, the high recall ensures that the broken samples, which developers do not want to overlook, can be detected without being overlooked.

Among the different model implementations (i.e., LR, LGB, RF), none consistently demonstrated superior performance. The bar graph for repairs prediction models (Figure 4.5a) has short



(a) Repairs prediction models.



(b) Breaks prediction models.

Figure 4.5: Prediction performance of (a) repairs and (b) breaks prediction models for each repair method and dataset.

error bars, indicating that the performance is not significantly different regardless of the implementation. However, there is an exception when the repair method is CARE and the dataset is Credit or IMDB, where the accuracy can differ by approximately 0.3 depending on the implementation. On the other hand, the bar graph for breaks prediction models (Figure 4.5b) generally has longer error bars compared to repairs prediction models, indicating that it is more sensitive to the model implementation.

Answer to RQ2: Repairs prediction models demonstrate good performance without imbalanced predictions (PR-AUC ranging from 0.6 to 0.9) for the repair method that yields a repair history with less bias in the number of repaired and non-repaired samples (i.e., Apricot and Arachne). However, a tendency of low precision and high recall is observed for CARE, which yields a biased repair history (i.e., a large number of non-repaired samples for a small number

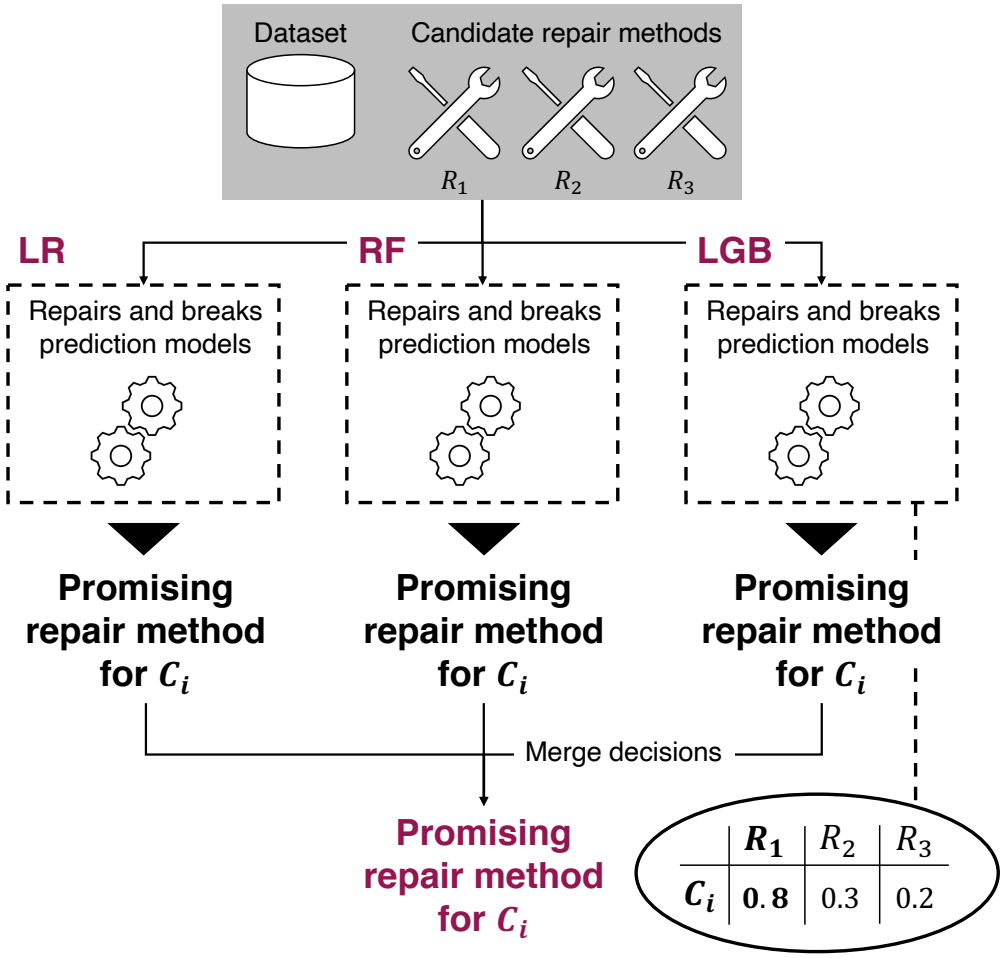


Figure 4.6: The flow of the experiment for RQ3. C_i denotes one of C_1, C_2, or C_3.

of repaired samples). Breaks prediction models tend to exhibit low precision and high recall across all repair methods.

6.3 RQ3: (Usefulness) Can repairs and breaks prediction models suggest a promising repair method in a shorter time compared to a naive method?

Approach. In this RQ, we address the problem of selecting the most promising repair method from a set of candidate repair methods. We demonstrate that the prediction models built in the previous RQ are effective in addressing the problem. Without any clues, we need to apply all the candidate methods, which takes a huge amount of time.

The most promising repair method depends on the situation in which it is applied. Therefore, we consider the following three indicators to determine how "promising" repair methods are:

- C_1: The ratio of repaired samples among misclassified samples.
- C_2: The ratio of non-broken samples among correctly classified samples.
- C_3: The difference between the ratio of repaired samples and the ratio of broken samples.

C_1 focuses on reducing misclassification, while C_2 focuses on maintaining correct classification. C_3 seeks a balance between them. The estimated values of these indicators can be computed by the prediction results of repairs and breaks prediction models. Based on these values, we suggest the most promising repair method from the perspective of C_i ($i = \{1, 2, 3\}$). If no promising method exists from the perspective, we suggest *No Operation (NOP)*, which means not to apply any repair method.

The selection flow for the most promising method is illustrated in Figure 4.6. Since there are three different implementations of the repairs and breaks prediction models (i.e., LR, RF, and LGB), we compute estimated values of C_i for each repair method in each implementation. Then, we obtain a promising method, which yields $\max(C_i)$ (the oval in the lower right of Figure 4.6), in each implementation. The exception is in the case of NOP, a user-defined lower bound th_{C_i} is set for each C_i to achieve NOP selection. If $\max(C_i) < th_{C_i}$, then NOP is output as there are no promising repair methods from the perspective C_i. The final decision is made by a majority voting on the outputs of the three model implementations. If a majority cannot be reached, the opinion of the model implementation with the highest prediction confidence is selected. The selected repair method is considered the most promising method (the lower center of Figure 4.6) and compared with the actual results of the repair methods to assess agreement.

We investigate the time savings that can be achieved by our method to demonstrate its usefulness in selecting a promising repair method in a shorter time. The baseline for comparison is the naive method, which applies all repair methods and selects the best one. We use this baseline to simulate the case where no clues, like our method, are available for selecting a promising repair method. We compare the total time for our method to the total time for the naive method.

- *The total time for our method* (T_{ours}): The sum of the execution times to select the promising repair method (T_{pred}) and run the selected repair method (T_{exe}), i.e., $T_{ours} = T_{pred} + T_{exe}$. The time required to select a repair method is the sum of the time required to train the repairs and breaks prediction models for the three repair methods and the time required for inference.
- *The total time for the naive method* (T_{naive}): The sum of the execution times of the three repair methods.

Table 4.7: The result of the selection of the most promising repair methods for each case in each dataset and the time saved by our method.

Type	Dataset	Case	Decision	Correct	T_{ours}	T_{pred}	T_{exe}	T_{naive}	%Saved time
Tabular	Credit	C_1	Apricot	✓	1m37s	0m46s	0m51s	1m17s	-26.75%
		C_2	NOP		0m46s	0m46s	0m00s	1m17s	-60.01%
		C_3	Apricot	✓	1m37s	0m46s	0m51s	1m17s	-26.75%
	Census	C_1	Arachne		11m19s	9m14s	2m04s	2h28m08s	-6.24%
		C_2	NOP		9m14s	9m14s	0m00s	2h28m08s	-6.24%
		C_3	Apricot	✓	2h32m18s	9m14s	2h23m03s	2h28m08s	-2.82%
	Bank	C_1	Apricot	✓	3h37m03s	12m05s	3h24m57s	3h32m15s	-2.27%
		C_2	NOP		12m05s	12m05s	0m00s	3h32m15s	-5.70%
		C_3	Apricot	✓	3h37m03s	12m05s	3h24m57s	3h32m15s	-2.27%
Image	FM	C_1	Apricot	✓	8h17m47s	3m54s	8h13m53s	10h6m50s	17.97%
		C_2	CARE	✓	1h08m36s	3m54s	1h4m42s	10h6m50s	88.69%
		C_3	Apricot	✓	8h17m47s	3m54s	8h13m53s	10h6m50s	17.97%
	C10	C_1	Apricot	✓	7h17m26s	6m46s	7h10m39s	9h11m09s	20.63%
		C_2	CARE	✓	34m50s	6m46s	28m03s	9h11m09s	93.68%
		C_3	NOP	✓	6m46s	6m46s	0m00s	9h11m09s	98.77%
	GTSRB	C_1	Apricot	✓	5h18m49s	2m42s	5h16m07s	6h14m48s	14.93%
		C_2	CARE		13m35s	2m42s	10m52s	6h14m48s	-0.72%
		C_3	Apricot	✓	5h18m49s	2m42s	5h16m07s	6h14m48s	14.93%
Text	IMDB	C_1	Apricot		9m37s	2m09s	7m27s	13m29s	-16.02%
		C_2	CARE	✓	2m27s	2m09s	0m18s	13m29s	81.71%
		C_3	CARE		2m27s	2m09s	0m18s	13m29s	-16.02%
	RTMR	C_1	Arachne	✓	18m10s	3m34s	14m36s	46m06s	60.56%
		C_2	Apricot		34m17s	3m34s	30m43s	46m06s	-7.74%
		C_3	Arachne	✓	18m10s	3m34s	14m36s	46m06s	60.56%
	Avg.			**66.7%** (16/24)					**16.29%**

Results. Table 4.7 presents the selected repair method for each case in each dataset and the time saved by our method. The Decision column represents the final decision that merges the results of the three model implementations. A check mark in the Correct column indicates that the selected repair method are consistent with the optimal repair method obtained from actual results. The %Saved time column shows ratio of time saved by our method to naive methods if the decision is correct, or the ratio of the overhead for selecting the repair method with a minus sign if the decision is not correct (because it is redundant time in that case). It should be noted that even if the decision is correct, the saved time can be negative if the overhead of the selection is large. A higher the value of this column indicates greater time savings from our method, while negative values indicate an increase in time. The thresholds for NOP selection were set to $th_{C_1} = 0.4$, $th_{C_2} = 0.8$, and $th_{C_3} = 0.2$.

By taking a majority vote of the LR, RF, and LGB opinions, the optimal repair method can be selected in 66.7% (16/24) of the cases. When choosing randomly among the four options (i.e., the three repair methods and NOP), the probability of selecting the optimal one is 25%, but using our method increases this probability. Out of 16 correct selections, 6 are for C_1, 3 for C_2, and 7 for C_3, indicating that these perspectives can be addressed without bias. For the image datasets (FM, C10, and GTSRB), which have long execution times for the repair methods, the optimal repair method is selected more often (in eight out of nine cases) compared to the other two types of datasets. This is because the characteristics of repaired or broken samples are more prominent in the image datasets, resulting in larger differences in the repair results between the methods, which are easier to distinguish. Our method is beneficial because it allows us to accurately select the optimal repair method when we need to select from repair methods with long execution times.

Our prediction model shows an average time savings of 16.29% compared to the naive method. Table 4.7 shows that our method saves time in eight out of nine cases for image datasets. Our method is beneficial because it provides significant time savings for image datasets where the repair execution time is long. In contrast, for tabular and text datasets, time savings are not observed in many cases because the execution time of the repair methods is shorter than the overhead of our method. For the three tabular datasets, saved times are negative in all nine cases, even though the promising repair methods are correctly selected in five out of nine cases. This is due to the particularly short execution time of the repair method for the tabular datasets (see Table 4.1). However, the overhead in the cases is only a few minutes to a dozen minutes, and the average time saving benefit is greater (i.e., average time saving is positive, 16.29%).

Our experiment in RQ3 shows that our method is effective when the candidate repair methods include a *time-consuming* repair execution. Hence, one possible use case for our method would be to select a promising method without having to apply all the time-consuming repair methods.

It is the practical case because the execution time of the repair method may be long due to factors such as the size of the dataset or model, or the type of the repair method.

Answer to RQ3: Our repairs and breaks prediction models can help select the most promising repair method without applying all repair methods. Specifically, the optimal repair method can be selected in the 16 out of 24 cases with little overhead. The average time savings of our method yields is 16.29%, which is useful for developers who want to apply the repair methods.

6.4 RQ4: (Transferability) Are repairs and breaks prediction models transferable to other repair methods?

Approach. In this RQ, we investigate how transferable our prediction models are to unknown repair methods. Building prediction models for future repair methods can be beneficial in selecting promising methods. However, it is time-consuming to obtain the application history of a repair method to build repairs and breaks prediction models. If we can transfer our existing prediction models to different repair methods, we can save time. Hence, our investigation focuses on assessing the extent to which our prediction model can be applied to unknown repair methods without compromising accuracy. This would be valuable for researchers and practitioners who intend to use our work in the future.

The flow of the experiment is shown in Figure 4.7. The pair (D, R) indicates the repair history of the dataset D and repair method R, on which the prediction model is trained. This is referred to as the *source*. The pair (D, R') indicates the target repair history to be predicted, which is referred to as the *target*. The only difference between the source and target repair histories is the target variable (i.e., the repair outcome). The dataset D is the same for both, so the explanatory variables for our prediction models are the same. Let m_{src} and m_{tgt} represent the prediction models trained on the source and target, respectively (i.e., $src = (D, R)$ and $tgt = (D, R')$). We evaluate the transferability of our prediction models from R to R' using two scores: *(1) Performance preservation* and *(2) Prediction consistency*.

(1) Performance preservation: This score indicates how close the prediction performance of m_{src} for the target is to that of m_{tgt}, which is the case when the prediction model trained on the target repair history is available.

$$\forall \text{Perf} \in P, \quad \text{Trans}_{\text{pp}}^{\text{Perf}}(src, tgt) = \frac{\text{Perf}(m_{src}, tgt)}{\text{Perf}(m_{tgt}, tgt)}. \tag{4.12}$$

Here, P is the set of the performance metrics, i.e., $P =$ {acc., pre., rec., f1., ROC-AUC, PR-AUC} and $\text{Perf}(m, D)$ indicates the performance (measured by Perf $\in P$) of the model m on the dataset D.

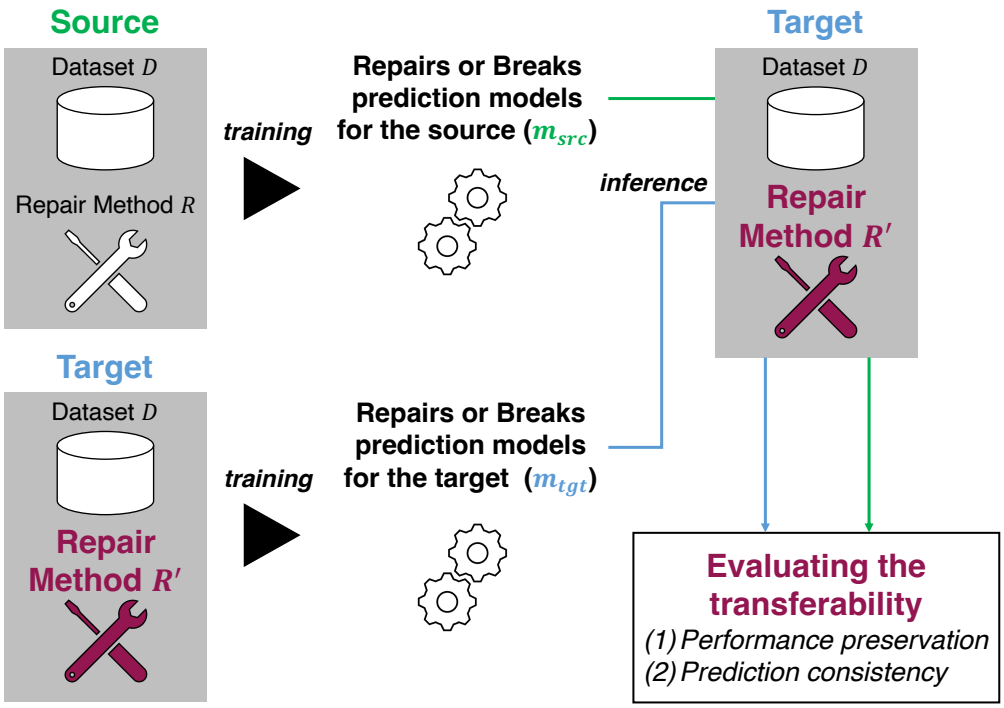


Figure 4.7: The flow of the experiment for RQ4.

If this score is less than one, it indicates a decrease in prediction performance due to the difference between the training source and the inference target. Conversely, if this score is one or more, it indicates a comparable performance compared to the case when the prediction model trained on the target repair history is available. In the paper, we report only the case Perf = f1 because this is the basic performance metric that takes into account the balance between precision and recall.[7] Thus, we simply denote $\text{Trans}_{\text{pp}}^{\text{f1}}(src, tgt)$ as $\text{Trans}_{\text{pp}}(src, tgt)$ in the paper.

This score focuses on the preservation of the prediction performance between the source and target. That is, the high score does not necessarily mean that the behavior (i.e., the prediction result for the same sample) matches between the source and target. The following score represents this point.

(2) Prediction consistency: This score indicates the degree of agreement of the prediction results between m_{src} and m_{tgt} for the target.

$$\text{Trans}_{\text{pc}}(src, tgt) = \frac{1}{|tgt|} \sum_{t \in tgt} m_{src}(t) = m_{tgt}(t). \tag{4.13}$$

The closer this score is to one, m_{src} behaves in the same way as m_{tgt} for the target. Even if the score defined in Equation 4.12 is high (i.e., the prediction performance is similar), it is possible that

[7] The results for other performance metrics can be checked in our replication package: https://github.com/posl/DNN-repair-break-prediction

each of m_{src} and m_{tgt} captures different patterns of repairs and breaks, so we have also added this score to the evaluation.

Note that the prediction consistency is symmetric for the source and target, but the performance preservation is not, i.e., $\text{Trans}_{\text{pc}}(src, tgt) = \text{Trans}_{\text{pc}}(tgt, src)$ and $\text{Trans}_{\text{pp}}(src, tgt) \neq \text{Trans}_{\text{pp}}(tgt, src)$. This difference is due to whether the target variable is used in the calculation of the score. Trans_{pp} includes the calculation of the prediction performance, which requires a target variable (different between the source and target). Therefore, it is not symmetric for the source and target. On the other hand, Trans_{pc} uses only explanatory variables (same between the source and target), so it is symmetric for the source and target.

Results. Tables 4.8 and 4.9 show the performance preservation (Trans_{pp}) and prediction consistency (Trans_{pc}) scores for each dataset, respectively. Since Trans_{pp} is not symmetric for the source and target, six patterns of scores are calculated for each dataset in Table 4.8. On the other hand, Trans_{pc} is symmetric for the source and target, so three patterns of scores are calculated for each dataset in Table 4.9.

The average performance preservation score is 0.75 for the repairs prediction model and 0.84 for the breaks prediction model. Notably, the scores for the repairs prediction models are lower than those for the breaks prediction models. This means that breaks prediction models are more transferable than repairs prediction models. This is because the broken sample had characteristics common to all repair methods, while the repaired sample had different characteristics for each method.

In both the repairs and breaks prediction models, Arachne has the highest average score when it becomes the inference target (i.e., 0.80 and 1.01, respectively, which can be seen in the second column from the right in the bottom row in Tables 4.8a and 4.8b). Interestingly, the average score for the breaks prediction model is 1.01, indicating that models trained on the history of repair methods other than Arachne perform slightly better than models trained on the history of Arachne. It is possible that predicting Arachne's repair history is easy because Arachne has characteristics that fall somewhere between CARE (conservative repair) and Apricot (disruptive repair through retraining). On the other hand, no repair method consistently shows high scores as the source.

The prediction models trained on the history of different repair methods capture different patterns of repairs and breaks. The average prediction consistency score is 0.69 for the repairs prediction model and 0.83 for the breaks prediction model, which is lower than the performance preservation score. This shows that even if the prediction performance is similar between the model trained on the source and the model trained on the target, their behavior may be different. In both Trans_{pp} and Trans_{pc}, it is common for the break prediction models to have higher scores

Table 4.8: The performance preservation ($Trans_{pp}$) of repairs and breaks prediction models. A dash ("-") indicates that the $Trans_{pp}$ is not calculated because the source and target are the same. "N/A" indicates that the $Trans_{pp}$ could not be calculated because the f1-score could not be calculated due to bias in the prediction results.

(a) For repairs prediction models.

Dataset	Src./Tgt.	CARE	Apricot	Arachne	*Avg.*
Credit	CARE	-	0.38	0.37	0.37
	Apricot	1.04	-	0.99	1.02
	Arachne	0.92	0.91	-	0.91
Census	CARE	-	0.83	0.76	0.79
	Apricot	0.95	-	0.92	0.93
	Arachne	0.63	0.87	-	0.75
Bank	CARE	-	0.81	0.80	0.80
	Apricot	0.94	-	0.97	0.96
	Arachne	0.93	1.01	-	0.97
FM	CARE	-	0.55	0.94	0.75
	Apricot	0.44	-	0.86	0.65
	Arachne	0.69	0.83	-	0.76
C10	CARE	-	0.62	1.03	0.82
	Apricot	0.39	-	0.65	0.52
	Arachne	0.75	0.94	-	0.84
GTSRB	CARE	-	0.36	1.31	0.83
	Apricot	0.45	-	0.46	0.46
	Arachne	0.91	0.47	-	0.69
IMDB	CARE	-	0.39	0.44	0.41
	Apricot	N/A	-	1.07	1.07
	Arachne	N/A	0.94	-	0.94
RTMR	CARE	-	0.33	0.28	0.30
	Apricot	N/A	-	0.95	0.95
	Arachne	N/A	0.99	-	0.99
Avg.		0.75	0.70	**0.80**	**0.75**

(b) For breaks prediction models.

Dataset	Src./Tgt.	CARE	Apricot	Arachne	*Avg.*
Credit	CARE	-	0.70	0.77	0.73
	Apricot	0.58	-	0.93	0.76
	Arachne	0.62	0.92	-	0.77
Census	CARE	-	0.97	0.85	0.91
	Apricot	1.03	-	0.83	0.93
	Arachne	0.75	0.80	-	0.78
Bank	CARE	-	0.95	0.96	0.96
	Apricot	1.01	-	0.98	1.00
	Arachne	0.80	0.80	-	0.80
FM	CARE	-	1.83	2.61	2.22
	Apricot	0.18	-	1.20	0.69
	Arachne	0.16	0.82	-	0.49
C10	CARE	-	0.33	1.25	0.79
	Apricot	0.19	-	0.71	0.45
	Arachne	0.28	0.59	-	0.44
GTSRB	CARE	-	1.77	2.32	2.05
	Apricot	0.36	-	0.34	0.35
	Arachne	0.47	1.44	-	0.96
IMDB	CARE	-	1.34	0.32	0.83
	Apricot	N/A	-	1.01	1.01
	Arachne	N/A	0.96	-	0.96
RTMR	CARE	-	0.87	0.23	0.55
	Apricot	N/A	-	0.81	0.81
	Arachne	N/A	0.58	-	0.58
Avg.		0.54	0.98	**1.01**	**0.84**

Table 4.9: The prediction consistency ($Trans_{pc}$) of repairs and breaks prediction models.

Dataset	Src. - Tgt.	repairs	breaks	Dataset	Src. - Tgt.	repairs	breaks
Credit	CARE - Apricot	0.52	0.65	C10	CARE - Apricot	0.73	0.68
	CARE - Arachne	0.60	0.56		CARE - Arachne	0.85	0.77
	Apricot - Arachne	0.85	0.83		Apricot - Arachne	0.53	0.57
Census	CARE - Apricot	0.76	0.96	GTSRB	CARE - Apricot	0.54	0.96
	CARE - Arachne	0.44	0.85		CARE - Arachne	0.92	0.98
	Apricot - Arachne	0.46	0.83		Apricot - Arachne	0.60	0.92
Bank	CARE - Apricot	0.72	0.96	IMDB	CARE - Apricot	0.64	0.81
	CARE - Arachne	0.72	0.89		CARE - Arachne	0.60	0.76
	Apricot - Arachne	0.95	0.90		Apricot - Arachne	0.89	0.86
FM	CARE - Apricot	0.67	0.89	RTMR	CARE - Apricot	0.59	0.89
	CARE - Arachne	0.87	0.85		CARE - Arachne	0.49	0.74
	Apricot - Arachne	0.80	0.95		Apricot - Arachne	0.86	0.82
					Avg.	**0.69**	**0.83**

than the repair prediction models.

Answer to RQ4: Our prediction models are transferable to other unknown repair methods in terms of prediction performance. The repairs and breaks prediction model shows an average performance preservation score of 0.75 and 0.84, respectively. Therefore, even if the target repair history is not available, our prediction models can still show the same performance as when it is available. However, the prediction consistency score is not always high, so the behavior may be different from when the target repair history is available.

6.5 RQ5: (Generality) Can the promising repair method selection be applied to DNN properties other than correctness?

Approach. Unlike previous RQs, this RQ applies our method to various properties, i.e., *robustness*, *fairness*, and *safety*. Table 4.10 shows the repair method applied to each property in our experiment. For the safety property, we only have the weights of the FNNs for ACAS Xu and no information about the training, such as the training dataset or the loss function used. This means that Apricot and Arachne, which require labeled datasets and the computation of loss functions, are not applicable with respect to the safety property. On the other hand, CARE and APRNN can

be applied to the safety property because they do not require labeled datasets and loss functions. Note that the safety property in our experiment can be verified without labeled datasets, as it is performed by randomly generating input values from a specific range and checking the output conditions. APRNN requires conditions on the output of DNNs as repair specifications, so we have only applied it to the safety property.

Table 4.10: The repair method applied to each property in our experiment.

Method	Correctness	*RQ5-1:* Robustness	*RQ5-2:* Fairness	*RQ5-3:* Safety
CARE	✓	✓	✓	✓
Apricot	✓	✓	✓	
Arachne	✓	✓	✓	
APRNN				✓

The experiment consists of the following three steps for each property studied in the corresponding sub-RQ.

(1) Obtaining repairs and breaks datasets. We use the definition of repaired and broken samples in Section 4.2 to obtain the repaired and broken datasets for each property (i.e., robustness, fairness, and safety). The characteristics of the repaired and broken samples for each property are illustrated by a radar plot similar to Figure 4.4 in RQ1.

(2) Building repairs and breaks prediction models. Using the repairs and breaks datasets obtained above, we build repairs and breaks prediction models in the same flow as in RQ2. We then evaluate the performance of these prediction models for each property.

(3) Selecting the promising repair method. Finally, we select the most promising repair method from the same three perspectives (i.e., C_1, C_2, and C_3) as in RQ3 and check how correct it is.

RQ5-1: Robustness.

We feed the FMC dataset into the original and repaired models for the FM dataset and obtain prediction results (same for C10). We build repairs and breaks datasets for both the FMC and C10C datasets by checking whether the predictions are correct before and after repair. These datasets are considered as repairs and breaks datasets in terms of robustness.

(1) Characteristics of repaired and broken samples. Figures 4.8a and 4.9a compare repaired and non-repaired samples, and broken and non-broken samples, respectively, in terms of robustness. Each radar plot shows the average metric values across all the datasets. For this property,

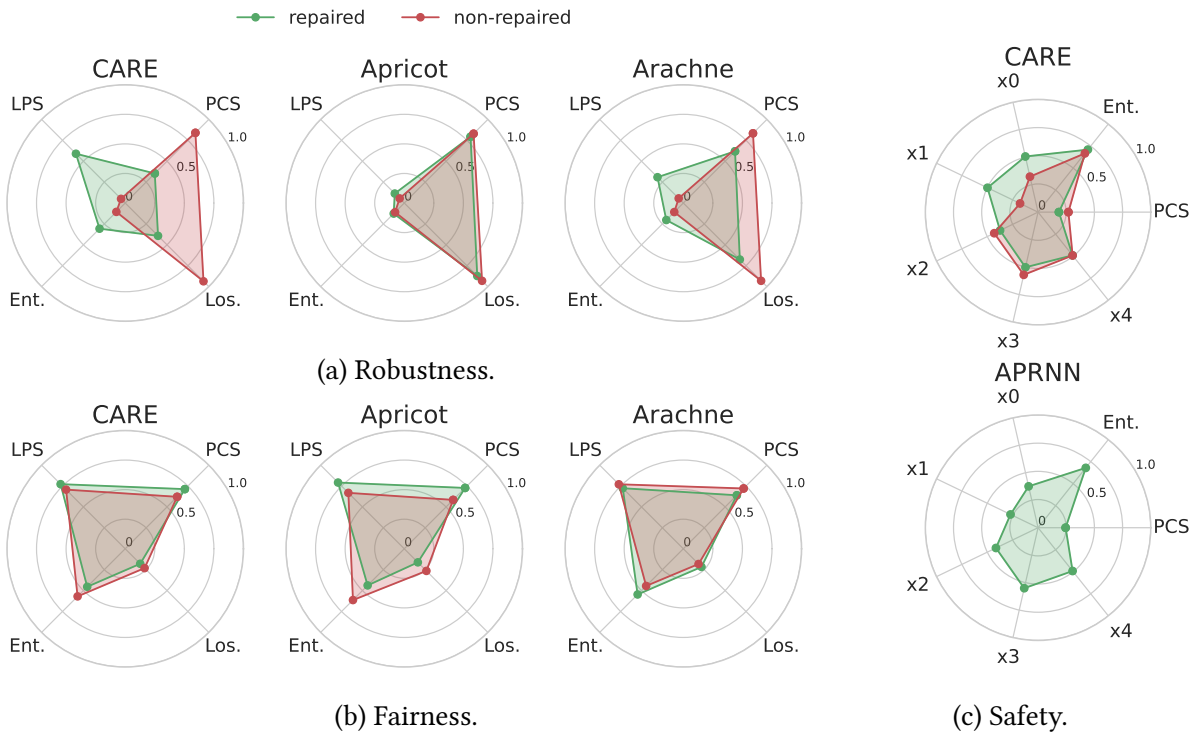
a similar tendency as for the correctness property (Figures 4.4a and 4.4b) is observed for both repaired and broken samples. Specifically, repaired samples tend to have high LPS and Entropy, while broken samples tend to have high Loss and Entropy. This is because the robustness, as defined in Section 4.2, is similar to the correctness in that it checks whether the predicted label is the same as the ground truth label.

(2) Performance of repairs and breaks prediction models. Tables 4.11 and 4.12 show the prediction performance of the repairs and breaks prediction models, respectively. The repairs and breaks prediction models for robustness exhibit comparable or slightly better performance than those for correctness. In detail, the acc. and rec. of the repairs prediction models, and the pre., rec., and PR-AUC of the break prediction models for robustness are higher than those for correctness.

(3) Selection of the promising method. Table 4.13a shows the results of the selection of the most promising methods in terms of robustness. The rows indicate the methods selected by our prediction models and the columns indicate the truth derived from the actual application results. For example, we see that there are six cases where Apricot was selected as the best by our method (the second row), two are correct and four are incorrect. There are 60 cases for selection ((19 datasets corresponding to each corruption of C10C + FMC) $\times$ 3 perspectives of C_1, C_2, C_3).

Our method selects the optimal repair method in terms of robustness in 55 out of 60 cases (91.7%). Out of these 55 cases, 33 cases were for NOP, 20 cases were for CARE, and 2 cases were for Apricot. The root cause is due to the actual distribution of optimal methods. The sum row represents the total number of cases where each repair method is optimal and the parenthesis represents in terms of correctness, obtained from RQ3. From this row, it can be observed that in terms of robustness, the optimal methods are concentrated on NOP and CARE, unlike in terms of correctness. Due to the difficulty to effectively repairing for robustness, NOP, and CARE which make conservative repairs, are often considered better. The results demonstrate that even in such cases, our prediction models can propose the most promising repair method with high accuracy. In other words, our prediction models can inform us in advance when it is better to do nothing or to make conservative updates, which is useful for avoiding unnecessary updates.

Answer to RQ5-1 (Robustness): (1) As with the case of correctness, repaired samples tend to have high LPS and Entropy while broken samples tend to have high Loss and Entropy. (2) The repairs and breaks prediction models for robustness exhibit comparable or slightly better performance than those for correctness. (3) Our method selects the optimal repair method in terms of robustness in 55 out of 60 cases (91.7%).



(a) Robustness.

(b) Fairness.

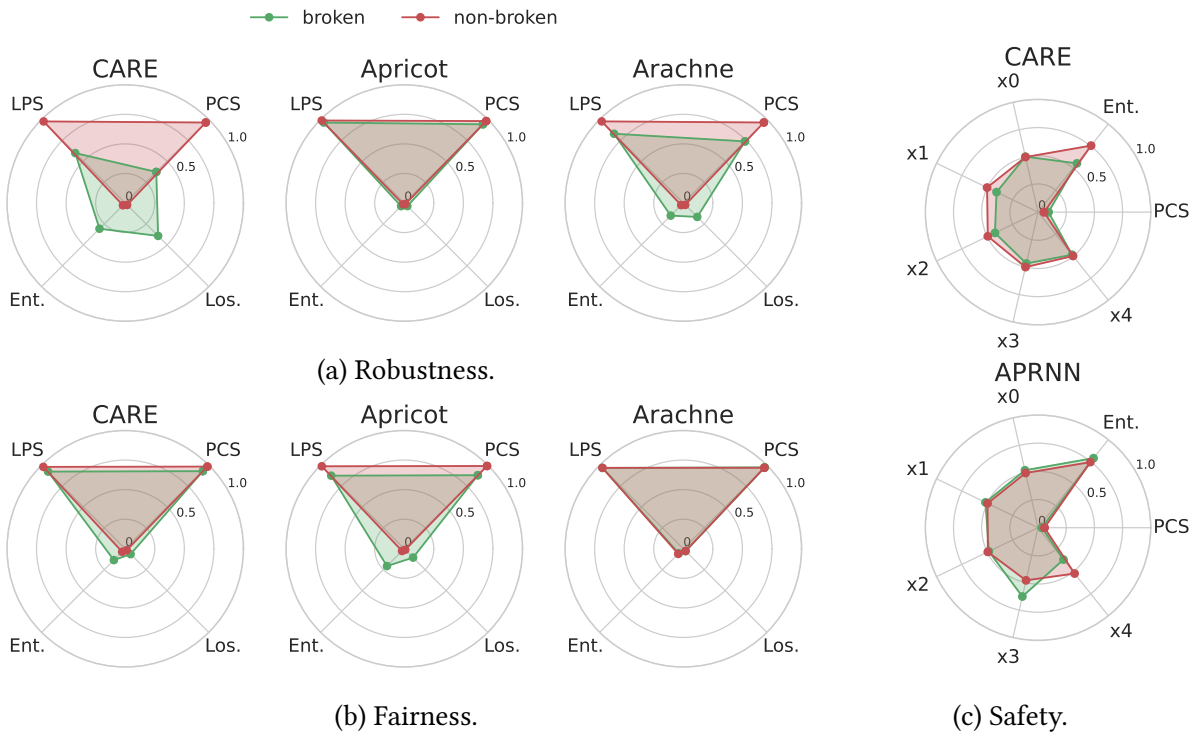
(c) Safety.

Figure 4.8: Comparison of the four metrics between repaired and non-repaired samples (robustness, fairness, and safety properties). Each radar plot represents the average of the four metrics for the repaired and non-repaired samples across all datasets.

RQ5-2: Fairness.

We use three tabular datasets (i.e., Credit, Census, and Bank) to evaluate our method in terms of fairness. For each data sample in each dataset, the values of the fairness property of the original and repaired models are calculated by Equation 4.8. We can obtain repairs and breaks datasets in terms of fairness by comparing these values as shown in Table 4.3.

(1) Characteristics of repaired and broken samples. Figures 4.8b and 4.9b compare repaired and non-repaired samples, and broken and non-broken samples, respectively, in terms of fairness. Overall, the differences between repaired and non-repaired (broken and non-broken) samples are small for all repair methods (i.e., a large overlap for the red and green radar in Figures 4.8b and 4.9b), which means that it may be difficult to distinguish between repaired and non-repaired (broken and non-broken) samples using four metrics. This tendency is different from that in terms of robustness, where a clear difference can be seen between the repaired and non-repaired (broken and non-broken) samples for the specific method, i.e., CARE. For the fairness property, the tendency of the metrics of the repaired and non-repaired samples is quite different from that for correctness and robustness (i.e., the shape of the radar plots in Figures 4.8a and 4.8b differs



(a) Robustness.

(b) Fairness.

(c) Safety.

Figure 4.9: Comparison of the four metrics between broken and non-broken samples (robustness, fairness, and safety properties). Each radar plot represents the average of the four metrics for the broken and non-broken samples across all datasets.

greatly.). For broken and non-broken samples shown in Figure 4.9b, broken samples have slightly higher Entropy and Loss, as is the case for correctness and robustness, but there is no significant tendency.

(2) Performance of repairs and breaks prediction models. From Tables 4.11 and 4.12, the acc. of the repairs and breaks prediction models in terms of fairness is lower than that in terms of correctness and robustness. Our results so far suggest that the four explanatory metrics may not be sufficient to accurately predict repairs and breaks in terms of fairness. The four metrics are based on the prediction probabilities output by the DNN. Therefore, they are effective for predicting repairs and breaks for properties such as correctness and robustness, which are related to whether the predicted label is the same as the ground truth label, but may not be effective for fairness, which does not depend on the correctness of the predictions.

(3) Selection of the promising method. Table 4.13b shows the results of the selection of the most promising methods in terms of fairness. There are nine cases for selection (three datasets (Credit, Census, and Bank) $\times$ three perspectives of C_1, C_2, C_3).

Our method selects the optimal repair method in terms of fairness in four out of nine cases

Table 4.11: Prediction performance of repair prediction models for each property and method. For each property and method, we show the average performance value across all datasets.

Property	Method	Acc.	Pre.	Rec.	PR-AUC
Correctness	CARE	0.772	0.127	0.511	0.261
	Apricot	0.695	0.705	0.729	0.736
	Arachne	0.684	0.482	0.705	0.568
	Avg.	**0.717**	**0.438**	**0.648**	**0.522**
Robustness	CARE	0.913	0.264	0.899	0.502
	Apricot	0.702	0.522	0.618	0.546
	Arachne	0.810	0.256	0.796	0.406
	Avg.	**0.808**	**0.347**	**0.771**	**0.485**
Fairness	CARE	0.568	0.548	0.559	0.585
	Apricot	0.600	0.726	0.619	0.755
	Arachne	0.556	0.774	0.593	0.781
	Avg.	**0.575**	**0.683**	**0.590**	**0.707**
Safety	CARE	0.848	0.735	0.881	0.817

(44.4%). This result is worse than the results for correctness and robustness, which shows the limitations of our method. This may be because the nature of fairness differs from correctness and robustness in our experiments. One of the future works is to use explanatory metrics that specialize in predicting repairs and breaks in terms of fairness.

Answer to RQ5-2 (Fairness): (1) The differences in the four metrics between repaired and non-repaired (broken and non-broken) samples are small for all repair methods. (2) The acc. of the repairs and breaks prediction models in terms of fairness is lower than that in terms of correctness and robustness. (3) Our method selects the optimal repair method in terms of fairness in four out of nine cases (44.4%), which shows the limitations of our method.

RQ5-3: Safety.

We obtain the repairs and breaks datasets for the safety property of the FNN models for ACAS Xu as described in Section 4.2. The data samples for safety property verification are randomly generated and do not have a ground truth label. This means that LPS and Loss, which require

Table 4.12: Prediction performance of break prediction models for each property and method. For each property and method, we show the average performance value across all datasets.

Property	Method	Acc.	Pre.	Rec.	PR-AUC
Correctness	CARE	0.888	0.044	0.634	0.304
	Apricot	0.756	0.239	0.684	0.343
	Arachne	0.748	0.266	0.633	0.366
	Avg.	**0.797**	**0.183**	**0.650**	**0.338**
Robustness	CARE	0.894	0.196	0.963	0.527
	Apricot	0.624	0.812	0.603	0.840
	Arachne	0.792	0.288	0.777	0.456
	Avg.	**0.770**	**0.432**	**0.781**	**0.608**
Fairness	CARE	0.689	0.324	0.666	0.389
	Apricot	0.739	0.458	0.708	0.595
	Arachne	0.621	0.495	0.663	0.503
	Avg.	**0.683**	**0.426**	**0.679**	**0.496**
Safety	CARE	0.810	0.255	0.823	0.313
	APRNN	0.888	0.711	0.929	0.830
	Avg.	**0.849**	**0.483**	**0.876**	**0.572**

a ground truth label, cannot be calculated. To compensate for the lack of metrics, we added the original five features (i.e., $x0, x1, x2, x3$, and $x4$) of ACAS Xu to the remaining two explanatory metrics (i.e., Entropy and PCS) for the repairs and breaks prediction models, resulting in a total of seven explanatory metrics.

Technically, we could add the original features of the sample itself as explanatory metrics for the other datasets as well. However, we did not do this for the other datasets because the repairs and breaks prediction models need to be lightweight. For example, in the case of image datasets, the number of the original features is 32x32=1024, which is too many to add to the explanatory metrics and increases the computational cost of the repairs and breaks prediction models.

(1) Characteristics of repaired and broken samples. Figures 4.8c and 4.9c compare repaired and non-repaired samples, and broken and non-broken samples, respectively, in terms of safety. We cannot obtain non-repaired samples for APRNN because it achieved a repair ratio of 100% (i.e., there is no red radar of APRNN in Figure 4.8c). These radar plots suggest that it may be useful to use the features of the data samples themselves to predict repairs and breaks. For example,

Table 4.13: Number of selected promising repair methods and actual optimal repair methods in terms of (a) robustness and (b) fairness. The number in the parenthesis in the sum row represents the total number of cases where each repair method is optimal in terms of correctness.

(a) Robustness.

Sel./Tru.	CARE	Apricot	Arachne	NOP
CARE	20	0	0	0
Apricot	0	2	0	4
Arachne	0	0	0	0
NOP	0	1	0	33
sum	20 (7)	3 (11)	0 (5)	37 (1)
	Correct	Total	Correct / Total	
	55	60	0.917	

(b) Fairness.

Sel./Tru.	CARE	Apricot	Arachne	NOP
CARE	0	1	0	0
Apricot	0	1	1	0
Arachne	0	0	3	0
NOP	2	1	0	0
sum	2 (7)	3 (11)	4 (5)	0 (1)
	Correct	Total	Correct / Total	
	4	9	0.444	

the repaired samples for CARE tend to have higher values for $x0$ and $x1$ than the non-repaired samples. In addition, the broken samples in APRNN tend to have higher values for $x3$ than the non-broken samples.

(2) Performance of repairs and breaks prediction models. We cannot build repairs prediction models for APRNN because non-repaired samples do not exist. Hence, we build repairs prediction models only for CARE. Breaks prediction models can be built for both CARE and APRNN. From Tables 4.11 and 4.12, both the repairs and breaks prediction models for safety show high acc. compared to those for other properties. This implies that using the features of the sample itself can contribute to a high accuracy.

(3) Selection of the promising method. Among the three indicators of how promising repair methods are, we cannot perform the promising method selection for C_1 and C_3 (repair ratio estimation is required) because repairs prediction models for APRNN are not available. Therefore, we only evaluate the selection results for C_2 (i.e., the ratio of non-broken samples among correctly classified samples), which can be estimated by the breaks prediction model. In cases where only the repair ratio is prioritized (e.g., C_1), even without creating a repairs prediction model, it is obvious that APRNN is the best method because it achieves a 100% repair ratio. However, the investigation only from the perspective of C_2 is still meaningful because it confirms whether our method works well when it needs to be applied.

Among the two FNNs $N_{2,9}$ and $N_{3,5}$ for ACAS Xu, we performed the promising repair method selection only for $N_{3,5}$. As a result, our method selected CARE as the most promising method in

terms of C_2, which is consistent with the actual repair results. For $N_{2,9}$, breaks prediction models for CARE are not available, so we were unable to perform selection.

Answer to RQ5-3 (Safety): (1) Repaired and non-repaired samples may differ in terms of the featurs of the data samples themselves. (2) Both the repairs and breaks prediction models for safety show high acc. compared to those for other properties. (3) For the safety property, although the cases where our method can be applied are limited, our method can correctly select the optimal method in such a case.

As for fairness and safety, it might be effective to use not only the explanatory metrics based on prediction probabilities but also the features of the sample itself and the features specific to each property. This means that our prediction model will lose its generality for properties, so further investigation is needed.

Summary of RQ5: Our promising repair method selection can be applied to the *robustness* property with the same or slightly better performance than the correctness property. For the *fairness* property, the performance of the selection is lower than that for the correctness and robustness, which shows the limitations of the four explanatory metrics. In terms of the *safety*, the cases where our method can be applied are limited due to the repair targets. However, our method works well when it can be applied.

7 Discussion

7.1 Use cases

In this section, we describe how our repairs and breaks prediction models can be used in the development and operation of real-world DNN systems. First of all, the main situation we consider is the case where repair histories are known for datasets during development. These histories may not be sufficient to determine which method is better for future data samples that are encountered during operation. Therefore, by applying a prediction model trained from the past history to samples encountered during operation, it is possible to predict the repair outcome by taking into account the characteristics of the new samples and the past repair history. Furthermore, our framework can be used to some extent even when the repair history is unknown, given the results of the transferability evaluation. In other words, one can apply our published repairs and breaks prediction models to unknown methods and obtain performance comparable to that obtained if

the history were known. Below are two use cases for our prediction models.

(1) Selection of promising repair methods. As shown in RQ3, our prediction models can select promising repair methods in a short time. This selection is positioned between DNN training and repair and is useful for the rapid development and operation of high-performance DNNs. Notably, our method is particularly effective when the candidate repair methods include a *time-consuming* repair execution, as the RQ3 results show (e.g., including an image dataset or a repair method whose execution time seems to be long or unknown). Even for tabular and text datasets, where the repair execution time is short, our method is useful for prioritizing the application of repair methods (i.e., deciding which method to try first). Additionally, developers can flexibly set the criteria for the promise of a repair method to meet their various needs. Our work triggers easy adoption of existing DNN repair techniques in practice.

(2) Prioritization of data samples to be verified during operation. By predicting samples that are likely to be repaired or broken, it is possible to prioritize data samples that should be verified during operation. For example, from a company's standpoint, there is strong resistance to breaking the normal behavior in previous versions of DNNs (i.e., regression) [177, 197]. Efficiently inspecting these possibilities with a limited budget is also important during DNN operations [48, 181, 185]. Potential regression candidates can be inspected in advance by using our breaks prediction models (e.g., for samples predicted likely to be broken, preferentially inspect which label they belong to, whether they are corrupt or not, and so on).

7.2 Limitations and Future Work

This section describes the limitations of our work and future research directions.

The applicability of our method to non-classification tasks such as regression (i.e., the prediction targets are not categorical but continuous numerical variables) and generation (e.g., machine translation and image generation) is unclear because our experimental targets are DNNs for classification tasks. This limitation is inherited from existing repair methods since they are designed for classification tasks. However, we believe that our method can be applied to these tasks by properly adapting the property functions defined in Section 4.2. Examples of such adaptations are shown below.

For regression tasks, a property function can be defined using the *mean squared error (MSE)* as follows:

$$f_M^{reg.}(x, y) = \begin{cases} 1 & (MSE(M(x), y) \leq \theta) \\ 0 & (MSE(M(x), y) > \theta) \end{cases}, \tag{4.14}$$

where $MSE(M(x), y)$ is the MSE between the predicted value $M(x)$ and the true value y, and θ

is the threshold of the desired performance of the model (note that smaller MSE is better). Based on this property function, we can identify the repaired and non-repaired (broken and non-broken as well) samples as shown in Table 4.3.

For the machine translation task, a type of generative task, a property function can be defined using the *BLEU score* [144] as follows:

$$f_M^{mt.}(x, Y) = \begin{cases} 1 & (BLEU(M(x), Y) \geq \theta) \\ 0 & (BLEU(M(x), Y) < \theta) \end{cases}, \tag{4.15}$$

where x is the original sentence, Y is the set of reference sentences, and $M(x)$ is the translated sentence by the model. Y corresponds to the ground truth in the labeled task and is needed to compute the BLEU score. This definition also uses the user-defined threshold θ to determine if the BLEU score is good enough or bad.

It is future work to investigate the applicability of our framework to these tasks. To investigate this, repair methods that can handle these tasks are needed.

7.3 Additional Analysis

Do different resampling strategies improve the performance of the repairs/breaks prediction model?

We performed RUS in RQ2 to reduce the samples of the majority class. In this section, we add two resampling strategies: without resampling (*w/o resampling*) and SMOTE [27] after RUS (*RUS+SMOTE*), and investigate the impact of different resampling strategies on prediction performance. In the case of w/o resampling, no resampling methods are performed, which means that the imbalance of the dataset is not addressed at all. On the other hand, RUS+SMOTE equalizes the number of samples of the majority and minority classes. This strategy first performs RUS to reduce the samples of the majority class as described in Section 6.2 and then performs SMOTE to increase the samples of the minority class to the same number as the majority class. We add this strategy because the original paper of SMOTE [27] reported that this combination achieves better performance than RUS alone for imbalanced datasets. We use the SMOTE implementation in the imbalanced-learn[8] Python package. We compare the prediction performance of the repairs and breaks prediction models with these three resampling strategies (i.e., RUS, RUS+SMOTE, and w/o resampling).

We show the performance of the repairs and breaks prediction model for different resampling strategies in Tables 4.14 and 4.15, respectively. The RUS column shows the prediction performance

[8] https://imbalanced-learn.org/stable/index.html

Table 4.14: The performance of the repairs prediction model for different resampling strategies.

		precision			recall		
Method	Dataset	RUS	RUS+SMOTE	w/o resampling	RUS	RUS+SMOTE	w/o resampling
CARE	Credit	0.009	0.012	**0.027**	0.333	0.333	**0.500**
	Census	0.019	0.017	**0.020**	**0.625**	0.524	0.540
	Bank	**0.019**	0.018	0.018	0.701	0.655	**0.705**
	FM	0.195	0.201	**0.214**	0.873	**0.915**	0.862
	C10	**0.374**	0.326	0.347	0.802	**0.893**	0.886
	GTSRB	0.403	0.360	0.403	0.756	**0.829**	0.764
	IMDB	0.000	0.000	0.000	0.000	0.000	0.000
	RTMR	0.000	0.000	0.000	0.000	0.000	0.000
Apricot	Credit	0.854	0.854	0.854	0.826	0.826	0.826
	Census	0.805	0.805	0.805	0.700	0.700	0.700
	Bank	0.849	0.849	0.849	0.792	0.792	0.792
	FM	0.752	0.752	0.752	0.682	0.682	0.682
	C10	0.811	0.818	0.818	**0.651**	0.647	0.647
	GTSRB	**0.857**	0.855	0.855	0.741	0.741	0.741
	IMDB	**0.346**	0.345	0.345	0.727	0.727	0.727
	RTMR	0.370	0.370	0.370	**0.711**	0.703	0.703
Arachne	Credit	0.611	0.611	0.611	0.731	0.731	0.731
	Census	0.642	0.636	0.642	**0.661**	0.623	0.658
	Bank	0.677	0.677	0.677	0.796	0.796	0.796
	FM	0.509	0.511	0.511	**0.741**	0.740	0.740
	C10	0.237	0.226	**0.247**	**0.685**	0.644	0.664
	GTSRB	0.048	0.048	**0.072**	0.722	0.611	0.722
	IMDB	0.535	0.537	0.537	**0.609**	0.604	0.604
	RTMR	0.596	0.598	0.598	**0.692**	0.690	0.690

of the model built in RQ2, and the remaining two columns show the prediction performance of the model with the additional resampling strategy. If one of the three resampling strategies provides a higher precision or recall value than the other two strategies, it is bolded. From these tables, we can see that the impact of different resampling strategies on prediction performance is negligible. In most cases, the change in precision and recall values is limited to a few percent. Furthermore, there is no resampling strategy that shows consistently high performance. Therefore, we believe that changing the resampling strategy will not affect the results after RQ2.

Table 4.15: The performance of the breaks prediction model for different resampling strategies.

		precision			recall		
Method	Dataset	RUS	RUS+SMOTE	w/o resampling	RUS	RUS+SMOTE	w/o resampling
CARE	Credit	**0.055**	0.009	0.033	0.417	0.417	0.333
	Census	0.014	0.013	0.014	0.837	0.805	**0.838**
	Bank	0.009	0.009	**0.010**	**0.839**	0.805	0.835
	FM	0.041	0.043	**0.044**	1.000	1.000	1.000
	C10	0.133	**0.149**	0.135	**0.979**	0.963	0.975
	GTSRB	**0.097**	0.096	0.088	1.000	1.000	1.000
	IMDB	0.000	0.000	0.000	0.000	0.000	0.000
	RTMR	0.000	0.000	0.000	0.000	0.000	0.000
Apricot	Credit	**0.182**	0.181	0.180	0.675	**0.684**	0.650
	Census	0.211	0.205	**0.212**	0.794	**0.800**	0.790
	Bank	**0.143**	0.139	0.138	0.848	0.843	**0.864**
	FM	0.136	0.136	0.136	0.827	0.827	**0.830**
	C10	0.975	0.975	0.975	0.394	**0.407**	0.394
	GTSRB	**0.155**	0.124	0.109	0.773	0.794	**0.861**
	IMDB	**0.022**	0.020	0.021	0.543	0.543	0.514
	RTMR	0.085	0.064	**0.086**	**0.618**	0.561	0.612
Arachne	Credit	0.417	0.417	0.417	0.672	0.672	0.672
	Census	0.608	0.609	0.608	0.744	0.739	0.744
	Bank	0.477	0.476	0.479	0.800	**0.801**	0.799
	FM	0.077	**0.081**	0.079	**0.806**	0.790	0.791
	C10	0.123	**0.131**	0.124	**0.582**	0.524	0.577
	GTSRB	0.013	**0.016**	0.013	0.841	**0.870**	0.841
	IMDB	**0.180**	0.179	0.179	0.289	0.289	0.289
	RTMR	0.232	0.233	0.233	0.335	0.342	0.342

Does training from the histories of multiple repair methods improve the performance of the repairs/breaks prediction model?

In our main analysis, we trained the repairs and breaks prediction models using the history of the single repair method. In the additional analysis, we investigate whether training from the histories of multiple repair methods can enhance the repairs and breaks prediction models. For each dataset, we train the repairs and breaks prediction model using repairs and breaks datasets that combine the histories of the two repair methods (R_{src_1} and R_{src_2}). This model can be written as $m_{src_1+src_2}$. We have three patterns of the combination because we use three repair methods (i.e., CARE, Apricot, Arachne). We evaluate its performance and the effect of the combination

Table 4.16: The prediction performance (f1-score) and transferability of the repairs prediction model trained form the combined repair history. R_{src_1} and R_{src_2} are repair methods used to train the model, and R_{tgt} is not used for training. "N/A" indicates that the f1-score could not be calculated due to bias in the prediction results.

Dataset	R_{src_1}	R_{src_2}	R_{tgt}	F1-score for R_{src_1}	F1-score for R_{src_2}	Transferability for R_{tgt}
Credit	CARE	Apricot	Arachne	0.022 (-17.2%)	0.833 (-0.0%)	0.988 (-0.6%)
	CARE	Arachne	Apricot	0.021 (-10.2%)	0.664 (-0.2%)	0.912 (0.6%)
	Apricot	Arachne	CARE	0.813 (-2.3%)	0.669 (0.5%)	0.814 (-21.8%)
Census	CARE	Apricot	Arachne	0.034 (-7.9%)	0.746 (-0.4%)	0.907 (-0.9%)
	CARE	Arachne	Apricot	0.026 (-27.1%)	0.632 (-2.7%)	0.914 (5.7%)
	Apricot	Arachne	CARE	0.759 (1.4%)	0.608 (-6.3%)	0.931 (-2.0%)
Bank	CARE	Apricot	Arachne	0.035 (-6.8%)	0.818 (-0.1%)	0.969 (-0.1%)
	CARE	Arachne	Apricot	0.034 (-7.8%)	0.733 (0.2%)	1.019 (0.5%)
	Apricot	Arachne	CARE	0.828 (1.1%)	0.723 (-1.1%)	0.931 (-1.4%)
FM	CARE	Apricot	Arachne	0.144 (-54.3%)	0.708 (-0.4%)	0.872 (-7.4%)
	CARE	Arachne	Apricot	0.235 (-25.9%)	0.611 (1.2%)	0.785 (-5.7%)
	Apricot	Arachne	CARE	0.678 (-4.5%)	0.553 (-8.4%)	0.508 (-26.2%)
C10	CARE	Apricot	Arachne	0.503 (0.4%)	0.730 (1.6%)	1.076 (4.4%)
	CARE	Arachne	Apricot	0.431 (-13.5%)	0.373 (6.2%)	0.936 (-0.3%)
	Apricot	Arachne	CARE	0.723 (0.5%)	0.350 (-0.1%)	0.822 (10.1%)
GTSRB	CARE	Apricot	Arachne	0.312 (-39.8%)	0.769 (-2.6%)	0.603 (-54.0%)
	CARE	Arachne	Apricot	0.497 (-4.3%)	0.099 (13.8%)	0.405 (-14.6%)
	Apricot	Arachne	CARE	0.779 (-1.3%)	0.040 (-49.6%)	0.476 (-47.8%)
IMDB	CARE	Apricot	Arachne	N/A	0.470 (0.3%)	1.044 (-2.6%)
	CARE	Arachne	Apricot	N/A	0.587 (4.0%)	0.991 (5.1%)
	Apricot	Arachne	CARE	0.475 (1.3%)	0.602 (7.1%)	N/A
RTMR	CARE	Apricot	Arachne	N/A	0.476 (-1.1%)	0.948 (-0.3%)
	CARE	Arachne	Apricot	N/A	0.627 (-0.3%)	1.005 (1.5%)
	Apricot	Arachne	CARE	0.477 (-0.8%)	0.629 (0.1%)	N/A
			Avg.	**0.488 (-6.3%)**		**0.857 (-7.2%)**

Table 4.17: The prediction performance (f1-score) and transferability of the breaks prediction model trained form the combined repair history. R_{src_1} and R_{src_2} are repair methods used to train the model, and R_{tgt} is not used for training. "N/A" indicates that the f1-score could not be calculated due to bias in the prediction results.

Dataset	R_{src_1}	R_{src_2}	R_{tgt}	F1-score for R_{src_1}	F1-score for R_{src_2}	Transferability for R_{tgt}
Credit	CARE	Apricot	Arachne	0.020 (-37.7%)	0.282 (-1.4%)	0.908 (-2.5%)
	CARE	Arachne	Apricot	0.019 (-34.8%)	0.517 (0.4%)	0.913 (-0.5%)
	Apricot	Arachne	CARE	0.276 (-3.4%)	0.499 (-2.6%)	0.599 (-4.1%)
Census	CARE	Apricot	Arachne	0.027 (1.3%)	0.331 (-0.8%)	0.844 (-0.2%)
	CARE	Arachne	Apricot	0.020 (-24.0%)	0.664 (-0.6%)	0.806 (-16.9%)
	Apricot	Arachne	CARE	0.274 (-17.8%)	0.659 (-1.3%)	0.784 (-24.1%)
Bank	CARE	Apricot	Arachne	0.018 (-1.7%)	0.233 (-4.7%)	0.989 (0.7%)
	CARE	Arachne	Apricot	0.015 (-21.1%)	0.598 (0.1%)	0.806 (-15.4%)
	Apricot	Arachne	CARE	0.197 (-19.2%)	0.598 (0.1%)	0.789 (-22.0%)
FM	CARE	Apricot	Arachne	0.011 (-81.3%)	0.227 (-1.0%)	1.200 (-53.9%)
	CARE	Arachne	Apricot	0.009 (-83.5%)	0.141 (4.4%)	0.866 (-52.7%)
	Apricot	Arachne	CARE	0.212 (-8.6%)	0.148 (10.7%)	0.172 (-6.0%)
C10	CARE	Apricot	Arachne	0.288 (19.7%)	0.714 (23.9%)	1.029 (-17.7%)
	CARE	Arachne	Apricot	0.096 (-58.4%)	0.225 (16.1%)	0.545 (-7.6%)
	Apricot	Arachne	CARE	0.738 (32.2%)	0.161 (-19.3%)	0.876 (208.9%)
GTSRB	CARE	Apricot	Arachne	0.030 (-78.7%)	0.187 (-23.6%)	0.483 (-79.2%)
	CARE	Arachne	Apricot	0.102 (-35.3%)	0.033 (34.7%)	1.472 (-16.9%)
	Apricot	Arachne	CARE	0.183 (-24.9%)	0.012 (-50.7%)	0.214 (-55.0%)
IMDB	CARE	Apricot	Arachne	N/A	0.042 (2.8%)	0.998 (-1.7%)
	CARE	Arachne	Apricot	N/A	0.213 (0.2%)	0.869 (-35.3%)
	Apricot	Arachne	CARE	0.036 (-15.4%)	0.217 (2.2%)	N/A
RTMR	CARE	Apricot	Arachne	N/A	0.152 (1.5%)	0.796 (-1.3%)
	CARE	Arachne	Apricot	N/A	0.267 (-0.3%)	0.561 (-35.5%)
	Apricot	Arachne	CARE	0.108 (-28.5%)	0.247 (-7.7%)	N/A
			Avg.	**0.221 (-13.4%)**		**0.796 (-10.9%)**

using the following indicators:

1. *Prediction performance for R_{src_1} and R_{src_2}:* For each repair method R_{src_1} and R_{src_2}, we evaluate the prediction performance of $m_{src_1+src_2}$ and compare this to that of the model trained with only the history of a corresponding single repair method (i.e., m_{src_1} and m_{src_2}). This allows us to evaluate whether $m_{src_1+src_2}$ can predict the repairs and breaks well for both R_{src_1} and R_{src_2}.

2. *Transferability for R_{tgt}:* We evaluate the transferability of $m_{src_1+src_2}$ to the target repair method R_{tgt} (i.e., the remaining repair method not used for training). We then compare this to the transferability of the model trained with only the history of a corresponding single repair method, i.e., $\text{Trans}_{\text{pp}}(src_1 + src_2, tgt)$ vs. $\text{Trans}_{\text{pp}}(src_1, tgt)$ and $\text{Trans}_{\text{pp}}(src_1 + src_2, tgt)$ vs. $\text{Trans}_{\text{pp}}(src_2, tgt)$. This allows us to evaluate whether combining repair histories provides greater transferability.

We show the prediction performance and transferability of the repairs and breaks prediction model trained from the combined repair histories in Tables 4.16 and 4.17, respectively. In these tables, we show only the f1-score as the prediction performance because the same trend is observed for the other performance metrics. The percentages in parentheses indicate the comparison with the model trained from a single repair history. A positive value indicates an improvement in performance, and a negative value indicates a decrease in performance. From these tables, we can see that combining the histories of multiple repair methods does not significantly improve the prediction performance or transferability of the repairs and breaks prediction models. In the case of the repairs prediction model shown in Table 4.16, the improvement is only a few percent, but the decrease can be up to around 50%. Therefore, we recommend training the repairs and breaks prediction model from the history of a single repair method.

8 Threats to Validity

8.1 Internal validity

Training and repairing DNNs involve randomness. In our work, we addressed this by dividing the training set into folds and applying repair methods to the DNN for each fold five times. Nevertheless, there are several hyperparameters in both DNN training (e.g., the number of training epochs, the kernel size of convolutional layer, etc) and repair methods (e.g., the number of reduced models in Apricot, the number of iterations of evolutionary computation in Arachne, etc), which were

manually determined in our experiment. These different hyperparameters can lead to different repair results, which may affect the results of RQs.

8.2 External validity

Although we have included a wide range of datasets (tabular, image, and text datasets) and models (FNNs, CNNs, and RNNs) in the experiments, our results may not apply to other datasets and models, such as video datasets and Transformer networks [178]. We could not find any studies on the repair methods specific to Transformer networks, and thus, we were unable to include them in this study. We believe that our method is easily applicable to Transformer networks because the explanatory metrics described in Section 3.3 can be computed for any neural network that performs classification. However, it remains unclear whether the RQ results can be generalized to Transformer networks since the impact of different DNN architectures on repair results has not been investigated. That is, different methods may have different architectural strengths. Therefore, their influence needs to be further investigated.

The other external validity is that the evaluation is limited to classification tasks. Section 7.2 describes how the property function allows our definition of repaired and broken samples to be extended for non-classification tasks. By applying the concept of property functions, our method could be applied to various tasks (e.g., regression and generation), but its specific effectiveness is unclear.

The large language models (LLMs) or large vision models (LVMs) are out of the scope of our study. Although our method is conceptually applicable to these large models as long as we can obtain the final probability vector, it may be difficult in practice due to long inference times or large memory consumption.

Our main focus are conventional DNNs like our experimental subject because they are also important in some practical cases (e.g., cyber-physical system, safety-critical system). Large models certainly perform well in various tasks. However, they are less controllable than smaller ones because they are often provided as a service (e.g., ChatGPT, GitHub Copilot, etc.), which means that it is difficult to operate a software system including them while repairing their unexpected behavior. Even when running large models in a local environment, there are problems such as high energy consumption and long latency [157, 191]. Also, even if the models already perform well, it is impossible to rule out the possibility that the DNN will exhibit unexpected behavior given the nature of machine learning. In such cases, DNN repair methods are useful as operational countermeasures. In fact, there is a case study [128] that investigates the feasibility of repair methods in the industry. Our study is significant in that we propose a framework for building a prediction

model to predict the repair outcomes with little overhead, in order to further boost the operation of software systems that incorporate DNNs.

8.3 Construct validity

In this study, we used four explanatory metrics (Entropy, PCS, LPS, and Loss) to distinguish repaired and broken samples; however, there is little evidence to suggest that these metrics are optimal. There might be other suitable metrics as the explanatory metrics of the samples. One example is the uncertainty in the model obtained by Monte-Carlo (MC) Dropout [51]. Another possibility is leveraging the features of the samples themselves. Especially for image and text datasets, we can leverage feature extraction techniques that has been studied in the ML community [134, 49]. However, it should be noted that increasing the number of explanatory metrics leads to an increase in the execution time for the repairs and breaks prediction model construction and inference.

Other threats to construct validity may exist in the experimental design. One such threat is the definition of repaired and broken samples. In Section 5.2, we defined repaired and broken samples based on the results of five repetitions of repair runs, but there are multiple ways to define these samples. Similarly, multiple thresholds are possible for the NOP selection in RQ3 and RQ5. Nevertheless, we have only experimented with fixed configurations; these could be threats.

9 Conclusion

We investigated repairs and breaks of repair methods for DNNs to bridge the gap between research on the repair methods and their practical use. We applied four representative repair methods (CARE, Apricot, Arachne, and APRNN) to FNNs, CNNs, and RNNs to investigate the characteristics of repaired and broken samples. The experimental results showed that there were statistically significant differences in confidence and ambiguity metrics for DNN predictions between repaired and non-repaired samples, and broken and non-broken samples, respectively. By exploiting these differences, we constructed prediction models (i.e., repairs and breaks prediction models) to distinguish between these samples. We showed that these prediction models can be used to select the most promising repair method that is likely to satisfy the developers' needs from among several repair methods in a short time. Specifically, our method was able to select the optimal repair method in about 66.7% of cases, with an average time reduction of 16.29% compared to the naive method. We also discussed the applicability of our method not only to correctness (i.e., whether the prediction is correct or not), but also to robustness.

Chapter 5

Evaluating Mutation-based Fault Localization for Quantum Programs

1 Introduction

Quantum computers fundamentally differ from classical computers by leveraging the principles of quantum mechanics to perform computations [132]. As quantum computing research advances, interest in *quantum programs* has grown [152]. A quantum program comprises a sequence of operations on quantum bits (*qubits*), where each operation is known as a *quantum gate*.

The process of creating and debugging quantum programs is inherently challenging, as it requires specialized knowledge of quantum computing. For example, qubits cannot be copied like classical bits due to the no-cloning theorem [132]. Moreover, not all developers are necessarily well-versed in quantum computing [162]. Consequently, such quantum-specific aspects lead to issues such as code smells [29] and technical debt [74].

In this study, we apply and evaluate *mutation-based fault localization (MBFL)* for quantum programs with the aim of enhancing debugging efficiency. Fault localization is a technique for identifying faults within a program [189]. An effective fault localization method can assist developers [92], especially those who are not experts in quantum computing [162].

MBFL leverages the results of mutation analysis for quantum programs. Mutation analysis generates *mutants*, systematically modified versions of a program. In classical programs, this technique has been used not only for testing but also for fault localization [123]. While numerous studies have explored mutation testing for quantum programs [11, 118, 50, 179], its potential for fault localization remains unclear. These studies use *quantum mutation operations* (e.g., deleting quantum gates), specifically designed for quantum programs. We believe that these operations

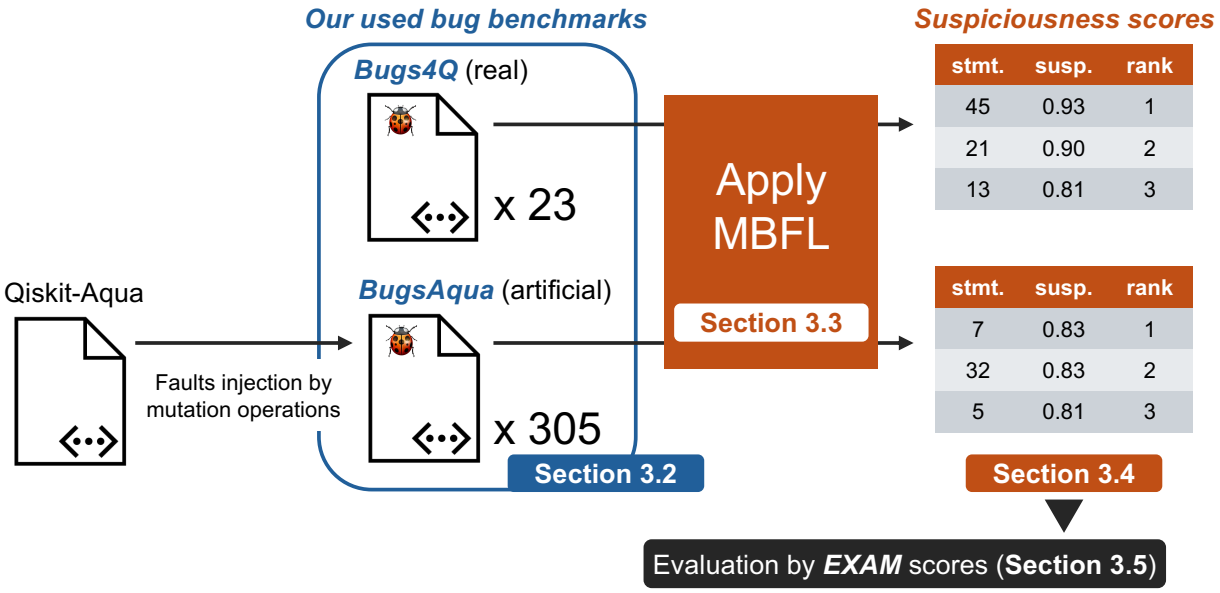


Figure 5.1: Our study design.

hold promise for enhancing fault localization in quantum programs.

Our evaluation involves 23 real-world faults and 305 artificial faults in quantum programs developed with Qiskit, a representative Python library for writing quantum programs. For mutation analysis of quantum programs, we use QMutpy [50][1], an extension of Mutpy, a mutation analysis tool for Python. The main contributions of this study are as follows:

- We demonstrate that real-world faults pose a greater challenge for MBFL in quantum programs than artificial faults in terms of EXAM score, which represents the percentage of the code examined before locating the faulty statement.
- We show that quantum mutation operations are effective for MBFL in quantum programs.
- We demonstrate the effectiveness of MBFL by comparing it with *spectrum-based fault localization (SBFL)*, a method that relies on execution path information.

2 Study Design

The goal of this study is to apply MBFL to quantum programs and evaluate its effectiveness. We select MBFL as the target technique for two reasons: (1) Mutation testing for quantum programs has been actively studied and MBFL is considered a promising approach for quantum programs.

[1] https://github.com/danielfobooss/mutpy

(2) MBFL has a broad scope of applicability, as it can be applied to quantum programs that include both classical and quantum instructions. Figure 5.1 shows our study design.

2.1 Research Questions

RQ1 ***(Artificial vs. real-world faults)*** *How does the performance of MBFL differ between artificial and real-world faults?*

RQ2 ***(Quantum vs. classical mutation operations)*** *Which is more effective, classical or quantum mutation operations?*

RQ1 evaluates MBFL for quantum programs in terms of the type of faults (i.e., artificial vs. real-world). It is similar to those in previous studies [145], which evaluate the fault localization methods for Java programs. On the other hand, RQ2 is specific to quantum programs. This RQ helps identify which types of mutation operations we should prioritize in our efforts.

2.2 Bug Benchmarks

We use both real-world and artificial bug benchmarks. For the real-world benchmark, we use *Bugs4Q*, proposed by Zhao et al. [214]. For the artificial benchmark, we create *BugsAqua*, which consists of bugs injected into Qiskit-Aqua[2] programs.

Real-world Benchmark (Bugs4Q)

Bugs4Q [214] is a benchmark consisting of 42 buggy programs written in Qiskit. These programs were collected from three popular platforms: GitHub, Stack Overflow, and Stack Exchange. For each buggy program, Bugs4Q provides the corresponding fixed version and test code to reproduce the bug. While the buggy and fixed programs were sourced from these platforms, the test code was manually written by the authors of the Bugs4Q paper. Each test code consists of a single test case. We use Bugs4Q because, to the best of our knowledge, it is the only benchmark that satisfies the following three criteria:

1. The buggy programs are written by developers, meaning the benchmark contains real-world bugs.
2. It includes both buggy and fixed versions of each program.
3. It provides test code to reproduce the bugs.

[2] https://github.com/qiskit-community/qiskit-aqua

The second and third criteria are essential for evaluating fault localization results. The test code is necessary to capture behavioral differences between the original program and its mutants, while the fixed program serves as a ground truth for faulty statements.

To reproduce the bugs in Bugs4Q, we cloned its replication package of Bugs4Q[3] and executed the buggy and fixed programs. For each sample in Bugs4Q, the test is expected to fail for the buggy version and pass for the fixed version. However, 19 out of 42 bugs could not be reproduced under our experimental setup. A possible reason is that the Bugs4Q paper and its replication package do not specify the versions of Python and related libraries, such as Qiskit, which may differ from our execution environment. As a result, we exclude these non-reproducible cases and use 23 samples (i.e., 23 buggy/fixed programs and their test code) in our study.

Artificial Benchmark (BugsAqua)

We create BugsAqua, an artificial bug benchmark, by injecting faults into Qiskit-Aqua programs. The motivation for using an artificial benchmark is twofold. First, since the number of the real-world bug benchmark is limited, incorporating an artificial bug benchmark allows us to evaluate the effectiveness of MBFL in a broader range of scenarios. Second, artificial and real-world bug benchmarks may have different characteristics, which could help highlight key factors essential for addressing real-world faults. We select Qiskit-Aqua as the target for fault injection because it has been used as an experimental target in the proposal paper for QMutPy [50], a mutation tool for quantum programs. Qiskit-Aqua contains quantum programs that implement typical quantum algorithms and includes extensive test cases written by developers of Qiskit.

Following their reproduction scripts[4], we selected the same 24 programs from the Qiskit-Aqua repository as their experimental targets. Each of these programs has a corresponding test code. On average, each test code contains 36.2 test cases, with a minimum of 1 and a maximum of 593 test cases, which is significantly more than those in Bugs4Q. We then apply QMutPy to these 24 programs, generating 2,361 mutants. Among them, 594 mutants have at least one failing test case, indicating that the faults are successfully injected. In this study, we exclude mutants whose execution time exceeds one hour. This one hour threshold corresponds to a level that satisfies approximately 10% of practitioners [92]. Although this level is relatively low, we set this timeout because a key focus of our study is to investigate the applicability and limitations of MBFL for quantum programs. Programs with an execution time exceeding one hour are deemed too time-consuming[5] for MBFL since it requires executing multiple mutants. In fact, the estimated time

[3] https://github.com/Z-928/Bugs4Q-Framework

[4] https://github.com/jose/qmutpy-experiments

[5] There were no cases in Bugs4Q where the execution time exceeded one hour.

for executing all the mutants can reach hundreds of hours in such cases. As a result, we use the remaining 305 mutants as an artificial bug benchmark. We refer to this benchmark as BugsAqua. BugsAqua meets the same two criteria as Bugs4Q: (1) A fixed version of the buggy program is available (by comparing it with the original code), and (2) The test cases are provided (written by Qiskit developers).

2.3 Mutating Buggy Programs

The first step in MBFL is to apply mutation operations to the buggy programs. We use QMutPy to generate mutants. In this study, we generate only first-order mutants, where a mutation operation is applied to a single statement in the program at a time. QMutPy supports 20 classical and 5 quantum mutation operations. We exclude two quantum mutation operations, quantum gate insertion and measurement insertion, because they do not work correctly for the programs in Bugs4Q.[6] As a result, we apply a total of 23 mutation operations to the buggy programs. QMutpy generated a total of 802 mutants for the 23 buggy programs in Bugs4Q and 34,627 mutants for the 305 buggy programs in BugsAqua.

The second step is to execute the test cases for the mutants. We use the test results for both the buggy program and its mutants to localize faulty statements in the buggy program.

2.4 Suspiciousness Scores

MBFL calculates the *suspiciousness score* for each statement based on the test results of the original program and its corresponding mutants. The suspiciousness score indicates the likelihood that the statement contains a fault. We use a simplified version of the formula proposed as MUSE [123], a representative MBFL method:

$$S(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \frac{|f_P(s) \cap p_m|}{|f_P(s)|}. \tag{5.1}$$

Here, $mut(s)$ represents the set of mutants generated by applying all mutation operations to statement s. $f_P(s)$ represents the set of failed test cases when statement s is executed in program P. Similarly, p_m represents the set of passed test cases in m, where m is a mutant of statement s. The suspiciousness score for s increases when mutating s frequently changes failing test cases to passing.

[6] This issue arises because identifying the insertion positions for quantum gates and measurements using the abstract syntax tree does not work properly.

2.5 Evaluation Metrics for Fault Localization

To evaluate the output of fault localization method (i.e., a ranked list of statements by suspiciousness scores in descending order), we use *EXAM* score, commonly used in existing studies [81, 145]. This score is defined as "the percentage of statements in a program that have to be examined until the first faulty statement is reached" [189]:

$$EXAM = \frac{\text{Rank of the faulty statement}}{\text{Total number of statements in a program}} \times 100\%. \tag{5.2}$$

The lower this percentage is, the more effective the method is, since it allows developers to find faulty statements with less effort.

In our experiments, the ground truth for the faulty statement is obtained from the fixed version of a buggy program in Bugs4Q and from the original version of a fault-injected program in BugsAqua, respectively. In BugsAqua, each buggy program contains only one faulty statement, whereas Bugs4Q may have multiple faulty statements, as it is a real-world bug benchmark. In the latter case, we use the rank of the highest-ranked faulty statement among the multiple faulty statements, as the EXAM score is defined based on the position of the "first" faulty statement.

Besides, there may be multiple statements with the same suspiciousness score as the faulty statement. In such cases, we adjust the calculation of the rank, which serves as the numerator in Equation (5.2). As suggested in a survey paper on fault localization [189], we report both the *best-case* and *worst-case* scenarios. In the best-case scenario, the faulty statement is assumed to be the first one found when checking all statements with the same suspiciousness score sequentially. Similarly, in the worst-case scenario, the faulty statement is assumed to be the last one found. If the rank of the faulty statement is r and the number of statements sharing the same suspiciousness score is n, then the rank used in the best-case and worst-case scenarios is r and $r + n - 1$, respectively.

2.6 Experimental Environment

All our experiments are performed on a classical computer. Although programs written in Qiskit can be executed on actual quantum computers, they can also be executed on classical computers as the quantum computer simulators. We used this simulator functionality because of the hurdles posed by the availability and noise of real quantum computers. The Python version used is 3.9.0. The versions of related to Qiskit are as follows: *qiskit-aer: 0.10.0*, *qiskit-aqua: 0.9.5*, *qiskit-ignis: 0.7.1*, *qiskit-terra: 0.20.0*.

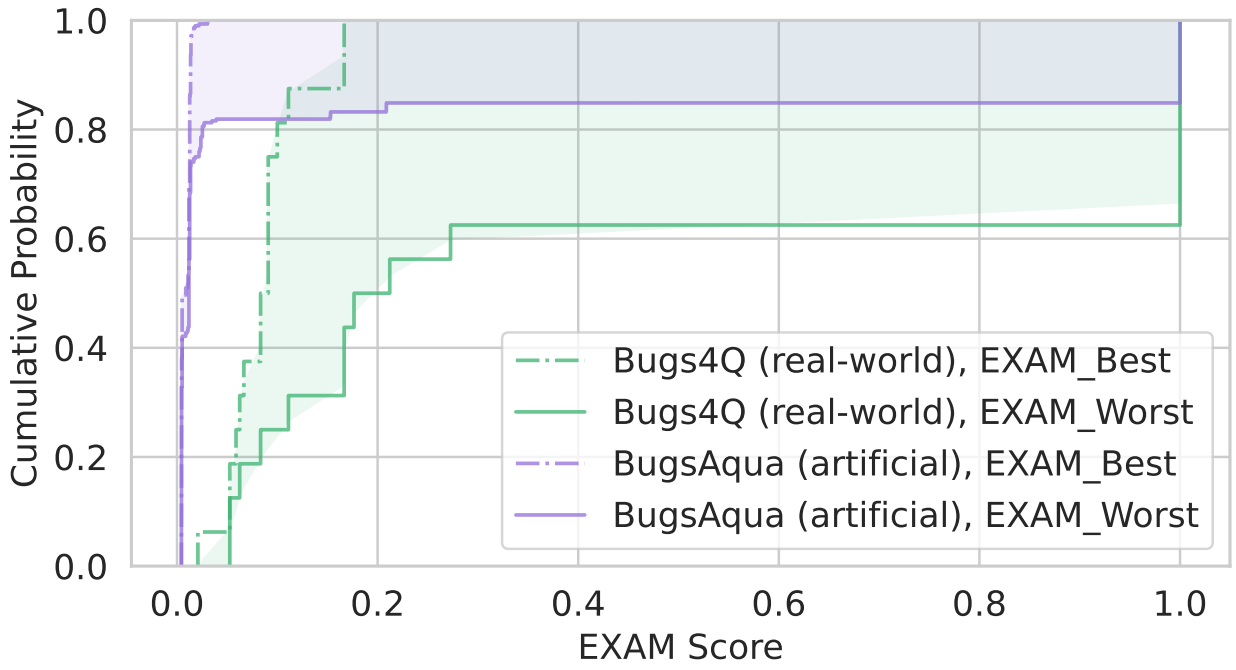


Figure 5.2: Empirical cumulative distribution function of EXAM scores for real-world faults (Bugs4Q) and artificial faults (BugsAqua). The filled areas indicate the range between the best-case and worst-case EXAM scores.

3 Results

3.1 RQ1: Artificial vs. Real-world Faults

Figure 5.2 presents the empirical cumulative distribution function of EXAM scores for Bugs4Q and BugsAqua. This figure indicates that real-world faults are more challenging for MBFL than artificial faults because the EXAM scores are higher for Bugs4Q than for BugsAqua. The median EXAM score in the best-case scenario is 8.7% for Bugs4Q and 0.9% for BugsAqua. In the worst-case scenario, the median EXAM score is 19.4% for Bugs4Q and 1.2% for BugsAqua. This finding is consistent with the results for Java program conducted by Pearson et al. [145]. One possible explanation for this result is the presence of "reversible" mutants, also reported in the study of Pearson et al. [145]. In our study, for example, an artificial fault injected by replacing a quantum gate can be fixed by applying a mutation operation that replaces it back with the original quantum gate. In contrast, real-world faults were less likely to be "reversible" through such simple mutation operations.

The range of EXAM scores for Bugs4Q is wider than that for BugsAqua (i.e., filled areas in Figure 5.2), indicating greater variability in MBFL performance for Bugs4Q. In Bugs4Q, the worst-

Table 5.1: Numbers of behavior-changing mutants (shown in the column #Mutants (B.-C.)). The column Avg. per Op. is calculated as #Mutants (B.-C.) divided by #Ops.

Benchmark	Mutation Type	#Ops.	#Mutants (B.-C.)	Avg. per Op.
Bugs4Q	Quantum	3	31 (93.9%)	10.3
	Classical	20	2 (6.1%)	0.1
BugsAqua	Quantum	3	200 (39.8%)	66.7
	Classical	20	302 (60.2%)	15.1

case EXAM score is 100% in approximately 40% of cases. This suggests that MBFL may fail in many cases. The reason for this can be that faults in Bugs4Q are too complex to detect with simple mutation operations as discussed in Section 4.1. Furthermore, test case quality may impact MBFL performance. Bugs4Q provides only one test case per bug, written by its authors rather than the original developers. These test cases may lack robustness to capture behavioral differences caused by mutants, potentially leading to poor results.

Answer to RQ1: Real-world faults are more challenging for MBFL than artificial faults. In the worst-case scenario, the median EXAM score is 19.4% for Bugs4Q and 1.2% for BugsAqua. Additionaly, the performance of MBFL for Bugs4Q is more unstable than that for BugsAqua.

3.2 RQ2: Quantum vs. Classical Mutation Operations

As shown in Equation (5.1), the success of MBFL depends on the presence of mutants that change test results (i.e., *behavior-changing mutants*). Therefore, we examine the number of behavior-changing mutants for each category of mutation operation (i.e., quantum or classical). This analysis helps us understand the effectiveness of quantum and classical mutation operations for MBFL.

Table 5.1 shows the numbers of behavior-changing mutants in Bugs4Q and BugsAqua. For Bugs4Q, most of the behavior-changing mutants are quantum mutants (31 out of 33 mutants, 93.9%). It indicates that for Bugs4Q, the success of MBFL heavily depends on the use of quantum mutation operations. On the other hand, for BugsAqua, a larger proportion of behavior-changing mutants are classical mutants (302 out of 502 mutants, 60.2%). This is because faults in BugsAqua, particularly those injected by classical mutation operations, are often reversible using classical mutation operations.

Per mutation operation, quantum mutation operations generate more behavior-changing mu-

tants than classical mutation operations. For Bugs4Q, quantum mutation operations generate 10.3 behavior-changing mutants per operation, while classical mutation operations generate only 0.1 per operation. A similar trend is observed for BugsAqua (see line for Avg. per Op. in Table 5.1).

Answer to RQ2: The results suggest that quantum mutation operations are more effective than classical mutation operations in changing test results. For Bugs4Q, 93.9% of the behavior-changing mutants are quantum mutants.

4 Discussion

4.1 Why are real-world faults in quantum programs challenging for MBFL?

In this section, we compare cases where MBFL succeeded and failed in Bugs4Q. This comparison allows us to discuss the boundaries of faults that existing quantum mutation operations can and cannot handle. Furthermore, we explore future directions for enhancing MBFL in quantum programs.

Listing 5.1 is an example where MBFL assigned the highest suspiciousness score to the faulty statement.

Listing 5.1: A code fragment from id=1 in Bugs4Q.

```
1 qc = QuantumCircuit(3)
2 qc.cx(0, 1, label='Label', ctrl_state=0)
3 qc.ccx(0, 1, 2, label='Label', ctrl_state=1) # This line causes an
    error.
```

This fault occurred because the developer incorrectly specified the arguments for the `ccx` gate. Such a fault can be detected using a mutation operation that deletes the quantum gate.

Listing 5.2 is an example where MBFL failed to assign a high suspiciousness score to the faulty statement.

Listing 5.2: A code fragment from id=39 in Bugs4Q.

```
1 qc = QuantumCircuit(4, 4)
2 # for i in range(4): # These lines are
3 #     qc.h(i)        # added in the fixed ver.
4 qc.cx(3, 1)
5 qc.cx(1, 0)
6 qc.cx(0, 1)
```

```
7  qc.ccx(3, 2, 1)
8  qc.cx(1, 2)
9  qc.cx(3, 2)
10 qc.measure(0, 0)
```

This fault occurred because the developer forgot to initialize all four qubits with the `h` gate. The `h` gate transforms the quantum state $|0\rangle$ into $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, creating a superposition of 0 and 1. This type of initialization is commonly used in many well-known quantum algorithms [132]. MBFL with existing mutation operations cannot handle this type of fault (i.e., *pattern-related faults*).

Two possible approaches to address this issue are (1) using *high-order mutants (HOM)*, which apply mutation operations at multiple locations simultaneously, and (2) introducing this type of initialization as a new mutation operation because it is a commonly used pattern. We consider (2) to be a promising direction because HOM introduce efficiency challenges, such as a significant increase in the number of mutants [77]. Collaborating with experts for quantum computing or application domains (e.g., chemistry) could help identify such types of frequent operation patterns. Alternatively, mining version control histories could reveal common operation patterns that developers make in practice.

Future directions: Enriching quantum mutation operations would be a next step toward effectively detecting real-world faults in quantum programs. For instance, identifying common operation patterns in quantum programs (e.g., initialization) and introducing them as new mutation operations could enhance the effectiveness of MBFL.

4.2 Comparison between MBFL and SBFL

In this section, we compare the performance of MBFL and SBFL for the quantum programs. Unlike MBFL, SBFL does not rely on mutants; it utilizes differences in execution paths. This comparison allows us to investigate the benefits of MBFL gained from utilizing mutants in addition to execution paths. We use two representative formulas for SBFL, i.e., Ochiai [8] and Tarantula [80]. A key difference from MBFL is that they do not use $mut(s)$ like Equation (5.1).

For Bugs4Q, SBFL cannot be applied because there is only one failing test case per bug. Since SBFL calculates suspiciousness scores based on differences in execution paths, it cannot be applied when there is only a single test case. MBFL can still be applied in such cases, showing its broader applicability compared to SBFL.

For BugsAqua, we compare the EXAM scores of MBFL and SBFL for each of the 305 buggy programs. Table 5.2 shows the comparison results of MBFL vs. SBFL$_{\text{Ochiai}}$ and MBFL vs. SBFL$_{\text{Tarantula}}$.

Table 5.2: Comparison of EXAM scores between MBFL and SBFL methods. In the "Sig. level" column, "**" and "-" indicate $p < 0.01$ and $p > 0.05$, respectively.

Comparison	EXAM	Cliff's δ	Sig. level
MBFL vs. $\text{SBFL}_{\text{Ochiai}}$	best	-0.0106	-
	worst	-0.7491	**
MBFL vs. $\text{SBFL}_{\text{Trantula}}$	best	0.0001	-
	worst	-0.7491	**

For each comparison, we conducted a Wilcoxon signed-rank test [35] on both the best-case and worst-case EXAM scores. Since we hypothesize that MBFL outperforms SBFL, we applied a one-sided test to determine whether MBFL yields lower EXAM scores. Additionally, we assessed the effect size using Cliff's δ [33]. In this case, a negative δ indicates the extent to which the EXAM score for MBFL is lower compared to that for SBFL.

From Table 5.2, we observe that in the best-case scenario, the difference between MBFL and SBFL is negligible. In contrast, in the worst-case scenario, MBFL tends to achieve a significantly lower EXAM score compared to SBFL. Specifically, both comparisons exhibit statistical significance with $p < 0.01$ and a large effect size according to a guideline by Kitchenham et al. [91]. According to the definition of the worst-case EXAM score, these results suggest that SBFL tends to assign the same suspiciousness score to a larger number of statements, including the faulty statement. It indicates the unstable performance of SBFL compared to MBFL.

Summary: For Bugs4Q, SBFL cannot be applied because there is only one failing test case per bug. For BugsAqua, the performance of MBFL is not significantly different from SBFL in the best-case scenario. However, in the worst-case scenario, MBFL achieves a lower EXAM score than SBFL, indicating the performance instability of SBFL.

5 Threats to Validity

Construct validity: Excluding programs in our study design (e.g., due to reproducibility issues or timeouts) may have introduced bias in the benchmarks. Furthermore, evaluating the performance of fault localization methods using only the EXAM score may not be sufficient. While we aimed for a comprehensive evaluation by considering both the best-case and worst-case scenarios, using additional evaluation metrics could provide more robust results.

Internal validity: We made minor modifications to the source code of QMutPy to make it compatible with MBFL. For example, QMutpy exits with an error if all tests for the program under test fail, as it is originally designed for mutation testing. In the case of MBFL, we would like the mutation operation to be applied even in such cases. The minor modifications were necessary to achieve this. While these changes were minimal, they may have inadvertently affected other parts of the tool.
External validity: Since we ran quantum programs on a simulator by Qiskit, our findings may not be generalizable to real quantum computers. Moreover, it is unclear whether our findings would generalize to quantum programs written in frameworks other than Qiskit or in programming languages other than Python.

6 Conclusion

This study evaluates the effectiveness of MBFL for quantum programs written in Qiskit using both real-world and artificial bug benchmarks. The results of RQ1 show that the EXAM score for BugsAqua is lower than that for Bugs4Q in both best-case and worst-case scenarios. This findings suggest that MBFL is more effective for artificial faults than for real-world faults. The results of RQ2 indicate that quantum mutation operations are more effective than classical mutation operations in changing test results. For Bugs4Q, 93.9% of the mutants that change test results are generated by quantum mutation operations. We also discuss the future directions by diving deeper into the successes and failures of MBFL. One possible direction is to enrich quantum mutation operations and improve the effectiveness of MBFL for quantum programs. It also remains unclear whether quantum mutation operations serve as good substitutes for real-world quantum bugs. Although this study proceeded under the assumption that this is true to some extent, investigating this point separately would further deepen our understanding of faults in quantum programs.

Chapter 6

Extended Work: Leveraging Mutation Analysis for LLM-based Repair of Quantum Programs

1 Introduction

Quantum computing has drawn significant attention for its potential to outperform classical machines on specific computational tasks [132]. Potential application areas include cryptography [17], chemical simulations [25], and machine learning [18], among others. To harness the computational power of quantum computers, developers must write *quantum programs* that specify sequences of operations on quantum bits (*qubits*), where each operation is called a *quantum gate*.

Similar to classical (non-quantum) programs, quantum programs also need to be debugged to ensure their correctness [125, 102]. However, debugging quantum programs poses distinctive challenges, as it involves quantum-specific concepts such as superposition and entanglement [162, 107, 203]. Indeed, these quantum-specific concepts lead to unique software engineering challenges, such as quantum-specific technical debt [139, 74] and code smells [29].

To assist developers in debugging quantum programs, researchers have proposed *automated program repair (APR)* techniques specifically designed for quantum software [103, 59, 173]. The goal of APR is to generate patches that modify buggy programs to pass all test cases in a given test suite [100]. Guo et al. [59] investigated the capability of ChatGPT [137] to repair quantum programs. Their results showed that ChatGPT could fix only 17% of complex bugs that involve implementing quantum algorithms. Tan et al. [173] and Li et al. [103] proposed APR techniques that

generate quantum gates as repair patches and insert them into quantum programs. The current state-of-the-art approach, HornBro [173], has been shown to fix more bugs than ChatGPT-based APR methods. However, this improvement comes at the cost of increased program complexity due to the insertion of additional gates. In one instance, HornBro added up to 249 additional gates while fixing a buggy program, negatively impacting the understandability and maintainability of the repaired code [29]. In summary, existing APR techniques for quantum programs suffer from either (1) low repair success rates or (2) poor understandability and maintainability of the repaired programs.

In this study, we aim to improve APR for quantum programs by incorporating insights from *mutation analysis* into a large language model (LLM)-assisted repair process. Mutation analysis evaluates how small changes to specific parts of a program affect its execution results. For quantum programs, mutation analysis combines classical mutation operators applicable to classical programs (e.g., replacing arithmetic operators) with quantum mutation operations (e.g., adding and deleting quantum gates) [50]. This approach has demonstrated its effectiveness for testing [50, 11, 118, 119]. In our experiments, each program is first analyzed using mutation analysis. The LLM is then given the buggy quantum program, its stack trace, and the mutation analysis results as input to generate *fixed code* along with a *natural language explanation* of the applied repair. Generating a natural language explanation alongside the fixed code helps developers understand the intent and rationale behind the repair, which cannot be easily inferred from the fixed code alone. We hypothesize that mutation analysis results can improve the repair success rate and help the model produce richer explanations, as they provide more detailed dynamic information than simple execution outputs, such as stack traces. The research questions (RQs) are as follows:

RQ1 ***(Repair success rate)*** *Can mutation analysis results improve the repair success rate of buggy quantum programs?*

RQ2 ***(Quality of explanation)*** *Can mutation analysis results improve the quality of explanations generated by LLMs?*

Our experimental setup includes 18 real-world buggy quantum programs from the Bugs4Q dataset [214]. All of these programs are written in Python using Qiskit [76], a widely used library for quantum programming. Throughout the experiment, we use GPT-5 [138], a state-of-the-art LLM, as the base model for generating the program repairs. For mutation analysis, we employ QMutPy [50], a mutation testing framework designed for quantum programs. We compared four prompt configurations to analyze how different contextual information affects repair performance. Our main contributions are as follows:

- We present the first empirical evidence that mutation analysis can serve as a valuable signal for improving LLM-based APR for quantum programs.
- We demonstrate that incorporating both dynamic runtime information and mutation analysis results into the prompt can yield a higher repair success rate.
- We show that incorporating mutation analysis results can improve the quality of LLM-generated explanations by making positional descriptions more accurate, complete, and concise.

2 Study Design

This section describes the design of our study, including the benchmark, prompt configurations, and LLMs. Figure 6.1 provides an overview of our experimental process. Our goal is to investigate how incorporating mutation analysis results into LLM-based repair affects the repair success rate and the quality of generated explanations. At a high level, for each buggy program, we prepare four different prompt configurations. The LLM is instructed to generate both the fixed code and an explanation of the fix regardless of the prompting configuration used.

2.1 Bug Benchmark

In this study, we use Bugs4Q [214], a benchmark of real-world bugs in quantum programs implemented using Qiskit [76]. It contains 42 buggy programs and their fixed versions, collected from GitHub, Stack Overflow, and Stack Exchange. We used Bugs4Q benchmark because it satisfied the following criteria:

1. It contains bugs found in real-world quantum programs written by human developers.
2. It includes test scripts to verify the behavior of both the buggy and fixed code.

The first criterion is necessary to compare the LLM's explanations of the generated patches with the developers' true intents behind the repairs. The second criterion is essential to perform mutation analysis and to verify whether the generated patches are correct.

Among the 42 programs in Bugs4Q, we selected 18 programs as our experimental targets. First, we cloned the replication package of Bugs4Q[1] and confirmed whether each bug could be reproduced. Bugs4Q bugs exhibit two types of symptoms: *Throw Exception (TE)* (e.g., an exception raised by the Qiskit library) and *Wrong Output (WO)* (e.g., an AssertionError triggered when the

[1] https://github.com/Z-928/Bugs4Q-Framework

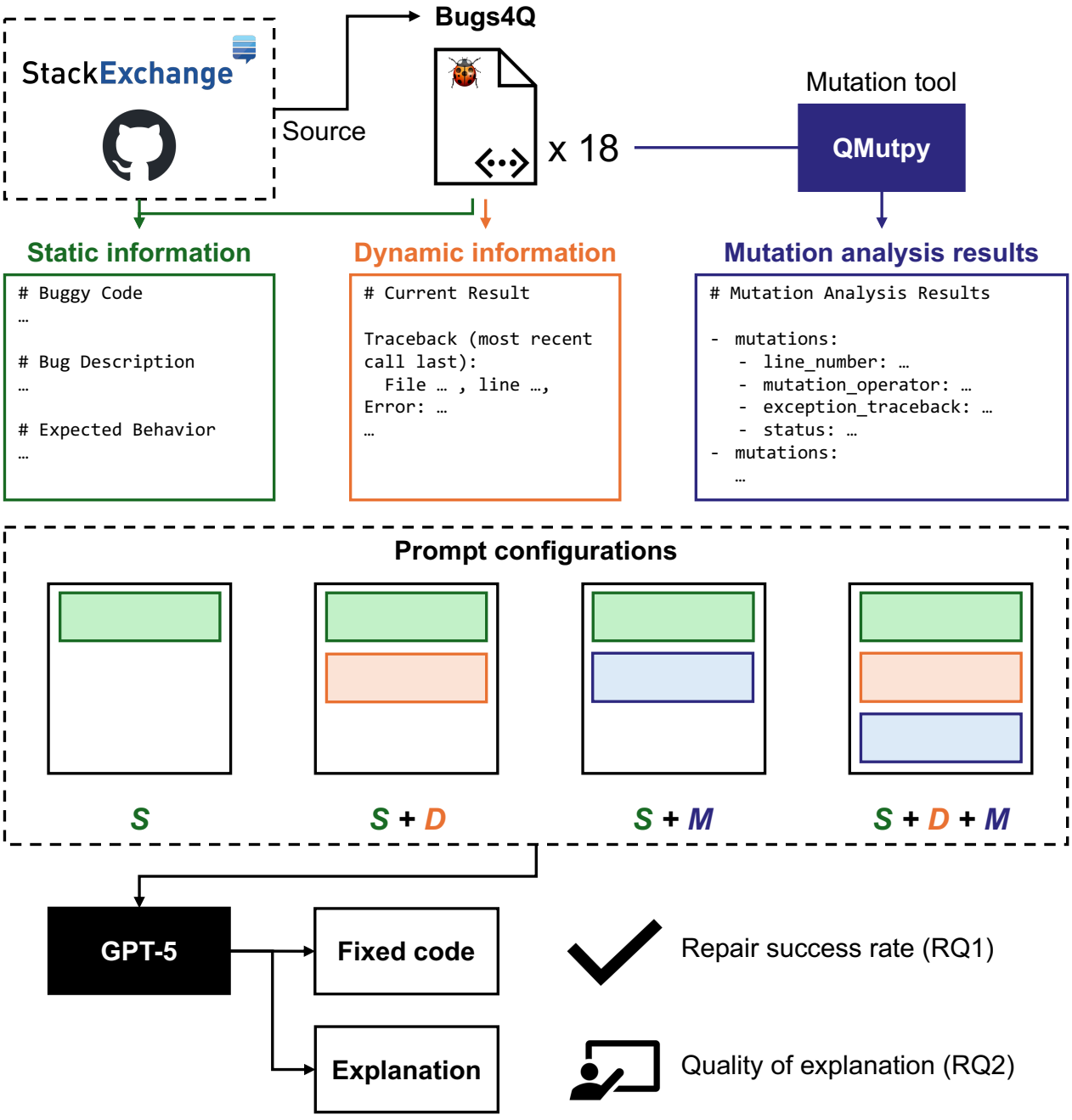


Figure 6.1: Overview of the experiment.

measured quantum state differs from the expected state). 19 programs were excluded because their bugs could not be reproduced. We also excluded 5 programs that did not satisfy either of the following criteria necessary to construct the prompts used in our experiment:

1. At least one mutant must be generated when performing mutation analysis.
2. The URL of the source repository must be accessible.

2.2 Prompt Construction

To investigate how the information included in prompts influences APR performance for quantum programs, we design the following four prompt configurations (see Figure 1.1):

- *S*: Static information only.
- *S+D*: Static and dynamic information.
- *S+M*: Static information and mutation analysis results.
- *S+D+M*: Static information, dynamic information, and mutation analysis results.

We describe how each type of information is obtained below.

Static Information. We collected three types of static information: (1) *buggy code* obtained from Bugs4Q; (2) *bug descriptions* obtained from the source URLs; and (3) *expected behavior* taken from the source URLs, describing the intended correct program behavior. (2) and (3) were manually collected by two authors, who inspected the source URLs of each bug and extracted the necessary information. Any disagreements were resolved through discussion to reach a consensus.

Dynamic Information. Dynamic information consists of the execution results of buggy programs, including the error message and the accompanying stack trace.

Mutation Analysis Results. We applied QMutPy [50] to each buggy program to obtain the mutation analysis results. QMutPy implements 20 classical and 5 quantum mutation operators, including insertion, deletion, and replacement of quantum gates, as well as insertion and deletion of quantum measurements. Each mutation operation is applied to a single location at a time. The mutation operators are applied repeatedly, producing multiple mutated programs (i.e., mutants) for each quantum program. The original program's test suite is executed on all mutants to determine their outcomes. We refer to the resulting collection of data as the mutation analysis results, which includes the following information for each applied mutation operation:

- *line_number:* The line number where the mutation operation was applied.
- *mutation_operator:* The name of the mutation operator applied to the program.
- *exception_traceback:* The traceback output when a test execution for the mutant fails.
- *status:* The result of the mutation test, categorized into one of the following: (1) *killed*, at least one test produces a different outcome; (2) *survived*, no change in test outcome; (3) *incompetent*, non-viable mutant (e.g., compile error, crash); (4) *time_out*, execution exceeded the time limit (e.g., infinite loop).

The system prompt is shared across all four prompt configurations. It provides detailed instructions on how to interpret and use each piece of information included in the prompt, as well as the strict output format to follow. All prompts used in our study are available in our replication package [13].

2.3 LLM

In this study, we used the state-of-the-art *GPT-5* via the OpenAI API with default settings. Because LLMs are stochastic and may produce different outputs for the same input, we generated outputs five times for each prompt configuration. Consequently, we obtained a total of 18 programs × 4 prompt configurations × 5 generations = 360 generated repairs and their corresponding explanations.

2.4 Evaluation Metrics

(RQ1) Repair Success Rate. We count a repair as successful if the generated code passes all the tests provided in Bugs4Q. The repair success rate is defined as the proportion of the 18 programs for which at least one of the five repair attempts was successful.
(RQ2) Quality of Explanation. The LLM-generated explanations may describe the cause of the bug, the location of the repair, and the rationale behind the repair. To evaluate these explanations, we adopt three core elements of patch explanations [105]: (1) *Position* — where the bug occurs; (2) *Cause* — why the bug occurs; (3) *Change* — how the bug is repaired. The quality of explanations is evaluated using three criteria proposed in previous work [129, 164]:

- *Correctness:* Is the explanation accurate with respect to the ground-truth fixed code?
- *Completeness:* Does the explanation fully describe all changes in the patch?
- *Complexity:* Does the explanation contain unnecessary complexity?

For each of the three explanation elements (Position, Cause, Change), we evaluate whether it satisfies each of the three criteria (Correctness, Completeness, Complexity) through a binary judgment (yes/no). For example, given an LLM-generated explanation, we evaluate the *Position* element by answering the following three questions: 1) does the explanation *correctly* explain the position of the bug?, 2) is the explanation *complete* and mentions all locations that were modified for the repair?, and 3) does the explanation include unnecessary *complexity* to explain the location of the bug?

Table 6.1: Repair success rates by prompt configuration and bug type. WO: Wrong Output, TE: Throw Exception. The maximum value in each column is highlighted in bold.

Prompt	Total [%]	WO [%]	TE [%]
S	77.8	70.0	**87.5**
S+D	88.9	90.0	**87.5**
S+M	83.3	80.0	**87.5**
S+D+M	**94.4**	**100.0**	**87.5**

Evaluating the three elements of patch explanation (Position, Cause, Change) therefore resulted in 9 (3 elements × 3 criteria) total binary evaluations per explanation generated by the LLM. We only considered the first generated repair patch among the five produced for each prompt, resulting in a total of 72 explanations. Two authors independently assessed each explanation, resolving any disagreements through discussion.

3 Results

We ran all experiments on simulators on classical computers due to the limited availability of quantum hardware and the innate instability of quantum computing posing a challenge for the reproducibility of our experiment. Additional information, such as the library versions used, is available in our replication package [13].

3.1 RQ1: Can mutation analysis results improve the repair success rate of buggy quantum programs?

Table 6.1 shows the repair success rates for each prompt configuration and bug type. The "Total" column reports the repair success rate across all 18 programs, while the "TE" and "WO" columns show those for each bug type (eight and ten programs, respectively). Out of 18 programs, 17 of them could be repaired by at least one prompt configuration.

Mutation analysis results are most effective when used in combination with dynamic information. Among all prompt configurations, *S+D+M* achieved the highest repair success rate at 94.4% in total. When considering only WO bugs, *S+D+M* successfully repaired all programs. Interestingly, *S+M* achieved a lower total repair success rate than *S+D*. Because mutation analysis requires running tests, the availability of *M* implies that *D* is also available. Thus, the lower success rate of *S+M* is not concerning, as *S+D+M* can easily be constructed in such cases. In contrast,

for TE bugs, the set of successfully repaired programs (seven out of eight) was identical across all prompt configurations. This indicates that dynamic information and mutation analysis results have no impact on buggy code in this category. For WO bugs, the program executes without crashing, making runtime information such as mutation analysis results helpful for repair. For TE bugs, the program terminates with an error, making the bug evident and static information alone sufficient.

Guo et al. [59] reported a repair success rate of 64% with their best prompt configuration using ChatGPT. Although we could not fully reproduce their execution environment and therefore used a different set of bugs, our results might surpass the existing state of the art. Two factors explain this improvement. The first is the advancement of LLMs: while Guo et al. [59] used GPT-3.5, our study used GPT-5. The second is our use of mutation analysis results to enrich the prompts. Based on the current results, it is not possible to clearly separate the contributions of these two factors. To disentangle them, additional experiments using multiple LLMs with different capabilities would be needed.

Answer to RQ1: Combining dynamic information with mutation analysis results is most effective, yielding a total repair success rate of 94.4%. It also achieved a 100% repair success rate for the bug type of the Wrong Output.

3.2 RQ2: Can mutation analysis results improve the quality of explanations generated by LLMs?

Table 6.2 reports how many of the 18 programs satisfy each criterion for each prompt configuration. The two evaluators agreed on 79.2% of the evaluations (513 out of 648 = 72 explanations × 9 evaluation items), and Cohen's κ [34] was 0.48, indicating moderate agreement [96]. This value is comparable to the $\kappa = 0.55$ reported in prior work [84] evaluating LLM-generated explanations for fault localization.

From the bolded values in Table 6.2, *S+D+M* achieved the best scores in six out of nine evaluation items, indicating that this prompt configuration produces better explanations compared to other prompts. In particular, for the position element, *S+D+M* achieved the best scores across all criteria, indicating that this prompt configuration is especially effective at positional explanations, leveraging dynamic information and mutation analysis results. On the other hand, *S+D+M* is less effective for the cause element. For this element, the best scores in correctness and completeness were achieved by the prompt configuration *S*. One possible reason is that most of the information relevant to the cause of the repair is already contained in the static informa-

Table 6.2: Quality of explanation for each prompt configuration. ↑ (↓) indicates that a higher (lower) value is better. The best value in each row is highlighted in bold.

Criterion	Element	*S*	*S+D*	*S+M*	*S+D+M*
Correctness (↑)	Position	12	12	13	**14**
	Cause	**14**	13	12	12
	Change	7	**9**	5	7
Completeness (↑)	Position	13	**15**	12	**15**
	Cause	**18**	15	16	15
	Change	13	**16**	13	**16**
Complexity (↓)	Position	2	2	**1**	**1**
	Cause	2	2	**1**	**1**
	Change	8	7	8	**6**

tion. Mutation analysis results are more effective for detailed positional explanations than for explaining the cause of the bug.

Answer to RQ2: Mutation analysis results combined with dynamic information achieved the best scores in six out of nine evaluation items by making positional descriptions more accurate, complete, and concise. On the other hand, their impact on explanation quality regarding the cause of the bug is limited.

4 Discussion

4.1 Unique Successful Repairs Across Prompt Configurations

Since RQ1 showed the effectiveness of dynamic information and mutation analysis results, we further investigate which programs particularly benefit from these components. Figure 6.2 shows the number of programs fixed by each prompt configuration. While all configurations repair 14 of the 18 buggy programs, the *S+D+M* configuration is the only one that succeeds in repairing 17 programs. This indicates that dynamic information and mutation analysis results did not hinder repair effectiveness, as the *S+D+M* configuration encompasses all successfully repaired programs. Our results demonstrate that mutation analysis can enhance the effectiveness of LLM-based APR

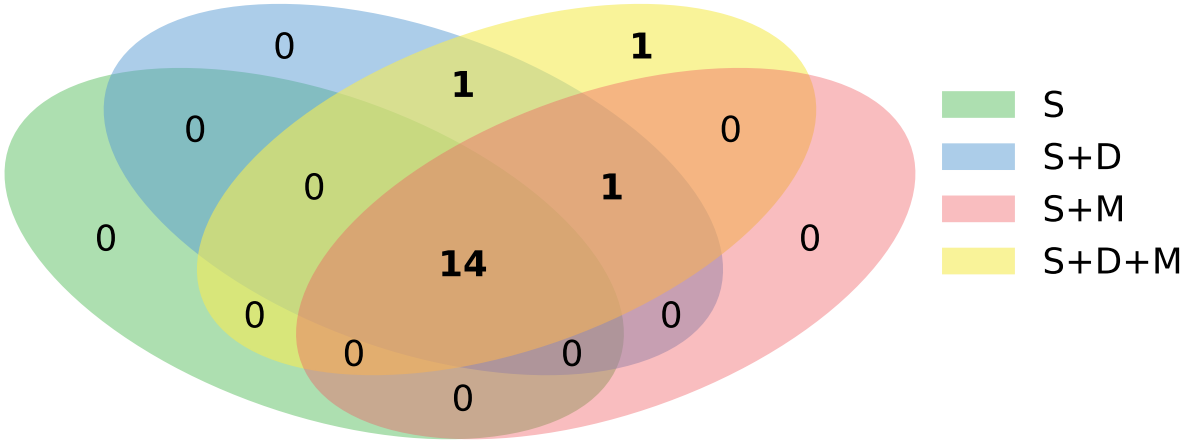


Figure 6.2: Successful repairs by prompt configuration.

for real-world buggy quantum programs, providing a concrete benefit.

4.2 Future Research Plan

(1) Exploring Additional Contextual Information. One limitation of our study is that it applies only to quantum programs for which mutation analysis can be performed. For three programs, we were unable to generate any mutants even though the mutation tool execution was successful. To address this limitation, we plan to incorporate additional types of contextual information (e.g., relevant sections of the Qiskit documentation) alongside mutation analysis results.

(2) Identifying Domain Knowledge Gaps. As another direction for future work, we plan to explore how LLM-generated explanations can help developers identify gaps in their quantum-specific knowledge. A prior survey shows that contributors to quantum open-source software (OSS) often lack sufficient understanding of physics and quantum computing [162]. Our workflow could help developers recognize such gaps through explanations generated during the repair. To validate the practical utility of this approach, we plan to conduct surveys with OSS developers. While LLM-based explanations of quantum algorithms have shown promise [38], explanations derived from program repair represent an unexplored avenue.

5 Threats to Validity

Construct Validity. In RQ1, we defined a successful repair as one that passes all tests, but this does not guarantee that the repair fully matches the developer's intended modification. This may affect the validity of the reported repair success rate.

Internal Validity. Due to the stochastic nature of GPT-5, the same prompt does not necessarily yield identical results. Although we generated five outputs for each prompt configuration, the inherent variability of LLM outputs may still influence the findings. In RQ2, two authors manually evaluated the quality of the explanations to mitigate subjectivity. However, some degree of subjective judgment is unavoidable, and misclassifications may have occurred due to limited domain knowledge in quantum programming.

External Validity. This study was conducted using quantum programs on the Qiskit simulator, and it is unclear whether similar results would be obtained on actual quantum computers. Since our investigation is limited to a single benchmark (Bugs4Q) and a single quantum framework (Qiskit), a full validation using a wider range of experimental subjects (i.e., different bug benchmarks, other frameworks, or LLMs other than GPT-5) is needed.

6 Conclusion

In this study, we demonstrated the effectiveness of incorporating mutation analysis results into prompts for LLM-based APR of quantum programs using real-world bug benchmarks. From our experiment, we find that the *S+D+M* prompt configuration, which includes static information, dynamic information, and mutation analysis results, achieves the highest repair success rate of 94.4%. We also find that explanations generated using *S+D+M* exhibit the lowest complexity while providing accurate, complete, and concise descriptions of bug positions. These results indicate that combining mutation analysis results with dynamic information can improve both the repair success rate and the quality of LLM-generated explanations. This work presents the first empirical evidence that mutation analysis can serve as valuable context information for improving LLM-based APR of quantum programs. This empirical evidence points toward new directions for developing APR techniques for quantum programs that enhance both reliability and explainability.

Chapter 7

Conclusion

This dissertation studied fault localization and repair techniques for AI-enabled and quantum software systems through four complementary studies, toward advancing the debugging of these emerging types of software systems. In summary, the contributions presented in this dissertation are as follows:

- In Chapter 3, we introduced PAFL, a novel fault localization approach for RNNs that abstracts RNNs as probabilistic automata and applies SBFL to sequences of state transitions. PAFL automatically identifies faulty state transition sequences on the probabilistic automata and extracts data samples that contribute to misbehavior by using these sequences. Our experimental results demonstrated that PAFL outperforms prior techniques such as RNNRepair in a majority of settings, achieving better performance in 66.74% of the cases. These results indicate that PAFL provides a practical direction for debugging AI-enabled systems that incorporate RNNs.
- In Chapter 4, we developed and evaluated the framework to predict the outcomes of repair methods for DNNs. Two types of prediction models were constructed: (1) predicting whether a misclassification can be corrected, and (2) predicting whether a correct classification may be turned into a misclassification by a repair method. Our experimental results demonstrated that these prediction models reduce the time required for selecting the most effective repair method, achieving an average time reduction of 16.29% compared to the naive approach of applying all repair methods and checking the results. This framework enables developers to effectively and efficiently apply existing DNN repair techniques to real-world AI-enabled software systems.
- In Chapter 5, we evaluated the effectiveness of MBFL for quantum programs. Using both real-world and artificial faults, we assessed how well MBFL can prioritize faulty statements.

We confirmed that real-world faults are more challenging for MBFL than artificial faults. In fact, the median EXAM score, which represents the percentage of the code examined before locating the faulty statement (lower is better), is 1.2% for artificial benchmark and 19.4% for the real-world benchmark. Additionally, the evaluation demonstrated that MBFL surpasses SBFL and that quantum-specific mutation operations are more effective than classical mutation operations. These results provide empirical support for MBFL as a viable technique for debugging quantum software systems.

- In Chapter 6, we investigated how mutation analysis results can enrich prompts for LLM-based repair of quantum programs. We instructed the LLM to produce both fixed code and a natural language explanation of the applied repair using mutation analysis results. Our evaluation showed that prompts augmented with mutation analysis results allowed the LLM to successfully repair 17 of the 18 faulty programs. In addition, the enriched prompts led to more accurate, complete, and concise positional descriptions in the generated explanations. These results show that combining mutation analysis with LLM-based repair yields a more effective debugging workflow for quantum software systems.

Rerefences

[1] Pytorch. https://pytorch.org/.

[2] Toxic comment classification challenge. https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge.

[3] Video shows tesla on autopilot slam into truck on taiwan highway. https://www.taiwannews.com.tw/en/news/3943199, 2020.

[4] 2 dead in tesla crash after car 'no one was driving' hits tree, authorities say. https://www.nbcnews.com/news/us-news/2-dead-tesla-crash-after-car-no-one-was-driving-n1264470, 2021.

[5] A. Abbas, A. Ambainis, B. Augustino, A. Bärtschi, H. Buhrman, C. Coffrin, G. Cortiana, V. Dunjko, D. J. Egger, B. G. Elmegreen, et al. Challenges and opportunities in quantum optimization. *Nature Reviews Physics*, pages 1–18, 2024.

[6] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[7] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46, 2006.

[8] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[9] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. Spectrum-based multiple fault localization. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.

[10] A. Alaboudi and T. D. LaToza. What constitutes debugging? an exploratory study of debugging episodes. *Empirical Software Engineering*, 28(5):117, 2023.

[11] S. Ali, P. Arcaini, X. Wang, and T. Yue. Assessing the effectiveness of input and output coverage criteria for testing quantum programs. In *Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation*, pages 13–23, 2021.

[12] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software engineering for machine learning: A case study. In *Proceedings of IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[13] Anonymous. Leveraging mutation analysis for llm-based automated quantum program repair, 2025.

[14] Anthropic. Claude. https://claude.com/, 2023. Accessed: 2025-11-04.

[15] T. Bai, J. Luo, J. Zhao, B. Wen, and Q. Wang. Recent advances in adversarial training for adversarial robustness. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, pages 4312–4321, 2021.

[16] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *Proceedings of the International Conference on Learning Representations*, 2017.

[17] C. H. Bennett, F. Bessette, G. Brassard, L. Salvail, and J. Smolin. Experimental quantum cryptography. *Journal of cryptology*, 5:3–28, 1992.

[18] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[19] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[20] M. Bonakdarpour, K. Campbell, D. Bonaglia, C. Rousseau, and S. Wunsch-Vincent. Global software spending surges to close to usd 700 billion in 2024, up 50% from 2020; the united states extends its lead. https://www.wipo.int/en/web/global-innovation-index/w/blogs/2025/global-software-spending, 2025. World Intellectual Property Organization (WIPO) Blog, June 2, 2025. Accessed: 2025-11-05.

[21] I. Bouzenia, P. Devanbu, and M. Pradel. Repairagent: An autonomous, llm-based agent for program repair. In *Proceedings of IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 694–694. IEEE Computer Society, 2025.

[22] G. W. Brier. Verification of forecasts expressed in terms of probability. *Monthly weather review*, 78(1):1–3, 1950.

[23] D. L. Calsi, M. Duran, X.-Y. Zhang, P. Arcaini, and F. Ishikawa. Distributed repair of deep neural networks. In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation*, pages 83–94, 2023.

[24] J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S.-C. Cheung. Deepfd: Automated fault diagnosis and localization for deep learning programs. In *Proceedings of the IEEE/ACM 44the International Conference on Software Engineering*, pages 573–585, 2022.

[25] Y. Cao, J. Romero, J. P. Olson, M. Degroote, P. D. Johnson, M. Kieferová, I. D. Kivlichan, T. Menke, B. Peropadre, N. P. Sawaya, et al. Quantum chemistry in the age of quantum computing. *Chemical reviews*, 119(19):10856–10915, 2019.

[26] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, pages 39–57, 2017.

[27] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[28] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[29] Q. Chen, R. Câmara, J. Campos, A. Souto, and I. Ahmed. The smelly eight: An empirical study on the prevalence of code smells in quantum computing. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*, pages 358–370, 2023.

[30] M. Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[31] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[32] Cirq Developers. Cirq, 2023.

[33] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

[34] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[35] W. J. Conover. *Practical nonparametric statistics*. john wiley & sons, 1999.

[36] A. I. Cowen-Rivers, W. Lyu, R. Tutunov, Z. Wang, A. Grosnit, R. R. Griffiths, A. M. Maraval, H. Jianye, J. Wang, J. Peters, et al. Hebo: Pushing the limits of sample-efficient hyper-parameter optimisation. *Journal of Artificial Intelligence Research*, 74:1269–1349, 2022.

[37] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight bug localization with ample. In *Proceedings of the sixth International Symposium on Automated analysis-driven debugging*, pages 99–104, 2005.

[38] G. d'Aloisio, S. Fortz, C. Hanna, D. Fortunato, A. Bensoussan, E. Mendiluze Usandizaga, and F. Sarro. Exploring llm-driven explanations for quantum algorithms. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 475–481, 2024.

[39] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. *ACM SIGSOFT Software Engineering Notes*, 21(3):121–134, 1996.

[40] G. Dong, J. Wang, X. Wang, and X. Wang. Towards interpreting recurrent neural networks through probabilistic abstraction. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 499–510, 2020.

[41] X. Du, Y. Li, X. Xie, L. Ma, Y. Liu, and J. Zhao. Marble : Model-based robustness analysis of stateful deep learning systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 423–435, 2020.

[42] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao. Deepstellar: Model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 477–487, 2019.

[43] R. Ehsani, E. Parra, S. Haiduc, and P. Chatterjee. Hierarchical knowledge injection for improving llm-based program repair. *arXiv preprint arXiv:2506.24015*, 2025.

[44] J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[45] H. F. Eniser, S. Gerasimou, and A. Sen. DeepFault: Fault localization for deep neural networks. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pages 171–191, 2019.

[46] S. Falkner, A. Klein, and F. Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the International Conference on Machine Learning*, pages 1437–1446, 2018.

[47] D. Feng, C. Haase-Schütz, L. Rosenbaum, H. Hertlein, C. Glaeser, F. Timm, W. Wiesbeck, and K. Dietmayer. Deep multi-modal object detection and semantic segmentation for autonomous driving: Datasets, methods, and challenges. *IEEE Transactions on Intelligent Transportation Systems*, 22(3):1341–1360, 2020.

[48] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 177–188, 2020.

[49] G. Forman et al. An extensive empirical study of feature selection metrics for text classification. *Journal of Machine Learning Research*, 3(Mar):1289–1305, 2003.

[50] D. Fortunato, J. Campos, and R. Abreu. Mutation testing of quantum programs: A case study with qiskit. *IEEE Transactions on Quantum Engineering*, 3:1–17, 2022.

[51] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the International Conference on Machine Learning*, pages 1050–1059, 2016.

[52] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019.

[53] A. Ghanbari, D.-G. Thomas, M. A. Arshad, and H. Rajan. Mutation-based fault localization of deep neural networks. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pages 1301–1313, 2023.

[54] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[55] Google. Gemini. https://gemini.google.com/, 2023. Accessed: 2025-11-04.

[56] Google. Google best practice, 2025. Accessed: 2025-11-13.

[57] D. Gopinath, K. Wang, M. Zhang, C. S. Pasareanu, and S. Khurshid. Symbolic execution for deep neural networks. *arXiv preprint arXiv:1807.10439*, 2018.

[58] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[59] X. Guo, J. Zhao, and P. Zhao. On repairing quantum programs using chatgpt. In *Proceedings of the 5th ACM/IEEE International Workshop on Quantum Software Engineering*, pages 9–16, 2024.

[60] 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*, volume 31, 2017.

[61] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[62] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[63] D. Hendrycks and T. Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. In *Proceedings of the International Conference on Learning Representations*, 2019.

[64] M. H. Hesamian, W. Jia, X. He, and P. Kennedy. Deep learning techniques for medical image segmentation: achievements and challenges. *Journal of Digital Imaging*, 32(4):582–596, 2019.

[65] H. Hofmann. Statlog (German Credit Data). UCI Machine Learning Repository, 1994. DOI: https://doi.org/10.24432/C5NC77.

[66] B.-J. Hou and Z.-H. Zhou. Learning with interpretable structure from gated rnn. *IEEE Transactions on Neural Networks and Learning Systems*, 31(7):2267–2279, 2020.

[67] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. Deepmutation++: A mutation testing framework for deep learning systems. In *Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1158–1161. IEEE, 2019.

[68] W. Huang, Y. Sun, X. Zhao, J. Sharp, W. Ruan, J. Meng, and X. Huang. Coverage-guided testing for recurrent neural networks. *IEEE Transactions on Reliability*, 2021.

[69] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks. In *Proceedings of the International Conference on Computer Aided Verification*, pages 3–29. Springer, 2017.

[70] Y. Huang and M. Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 541–553, 2019.

[71] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 1110–1121, 2020.

[72] Y. Ishimoto, M. Kondo, N. Ubayashi, and Y. Kamei. Pafl: Probabilistic automaton-based fault localization for recurrent neural networks. *Information and Software Technology*, 155:107117, 2023.

[73] Y. Ishimoto, K. Matsui, M. Kondo, N. Ubayashi, and Y. Kamei. An initial analysis of repair and side-effect prediction for neural networks. In *Proceedings of the IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI*, pages 80–85, 2023.

[74] Y. Ishimoto, Y. Nakamura, R. Katsube, N. Sato, H. Ogawa, M. Kondo, Y. Kamei, and N. Ubayashi. An empirical study on self-admitted technical debt in quantum software. In *Proceedings of the 31st Asia-Pacific Software Engineering Conference (APSEC)*, pages 41–50, 2024.

[75] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 510–520, 2019.

[76] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta. Quantum computing with Qiskit, 2024.

[77] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009.

[78] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM joint european*

software engineering conference and symposium on the foundations of software engineering, pages 1646–1656, 2023.

[79] M. S. Johnson. A software debugging glossary. *ACM Sigplan Notices*, 17(2):53–70, 1982.

[80] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.

[81] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao. Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software*, 90:3–17, 2014.

[82] K. D. Julian, M. J. Kochenderfer, and M. P. Owen. Deep neural network compression for aircraft collision avoidance systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019.

[83] 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*, pages 437–440, 2014.

[84] S. Kang, G. An, and S. Yoo. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1(FSE):1424–1446, 2024.

[85] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb. Maldozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, 24:S48–S59, 2018.

[86] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *Proceedings of the International Conference on Computer Aided Verification*, pages 97–117, 2017.

[87] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[88] J. Kim, R. Feldt, and S. Yoo. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 1039–1049. IEEE, 2019.

[89] J. Kim, N. Humbatova, G. Jahangirova, P. Tonella, and S. Yoo. Repairing dnn architecture: Are we there yet? In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation*, pages 234–245, 2023.

[90] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im. A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security*, 14(3):773–788, 2018.

[91] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong. Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, 22:579–630, 2017.

[92] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 165–176, 2016.

[93] R. Kohavi. Census Income. UCI Machine Learning Repository, 1996. DOI: https://doi.org/10.24432/C5GP7S.

[94] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. *Master's thesis, University of Tront*, 2009.

[95] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, 2002.

[96] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[97] 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 *Proceedings of 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[98] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2011.

[99] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[100] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[101] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[102] N. C. Leite Ramalho, H. Amario de Souza, and M. Lordello Chaim. Testing and debugging quantum programs: The road to 2030. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–46, 2025.

[103] Y. Li, H. Pei, L. Huang, B. Yin, and K.-Y. Cai. Automatic repair of quantum programs via unitary operation. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–43, 2024.

[104] D. Li Calsi, M. Duran, T. Laurent, X.-Y. Zhang, P. Arcaini, and F. Ishikawa. Adaptive search-based repair of deep neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1527–1536, 2023.

[105] J. Liang, Y. Hou, S. Zhou, J. Chen, Y. Xiong, and G. Huang. How to explain a patch: An empirical study of patch explanations in open source projects. In *Proceedings of the 2019 IEEE 30th International Symposium on Software Reliability Engineering*, pages 58–69, 2019.

[106] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 42:60–88, 2017.

[107] J. Luo, P. Zhao, Z. Miao, S. Lan, and J. Zhao. A comprehensive study of bug fixes in quantum programs. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 1239–1246. IEEE, 2022.

[108] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao. DeepCT: Tomographic Combinatorial Testing for Deep Learning Systems. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 614–618. IEEE, 2019.

[109] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang. DeepGauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, pages 120–131, 2018.

[110] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang. DeepMutation: Mutation Testing of Deep Learning Systems. In *Proceedings of the 29th*

International Symposium on Software Reliability Engineering (ISSRE), pages 100–111. IEEE, 2018.

[111] S. Ma, Y. Liu, W. C. Lee, X. Zhang, and A. Grama. Mode: Automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 175–186, 2018.

[112] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, 2011.

[113] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297, 1967.

[114] 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*, 18(1):50 – 60, 1947.

[115] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen. Learning probabilistic automata for model checking. In *Proceedings of the 8th International Conference on Quantitative Evaluation of Systems*, pages 111–120, 2011.

[116] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.

[117] V. Mavroeidis, K. Vishi, M. D. Zych, and A. Jøsang. The impact of quantum computing on present cryptography. *International Journal of Advanced Computer Science and Applications*, 9(3), 2018.

[118] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue. Muskit: A mutation analysis tool for quantum software testing. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, pages 1266–1270, 2021.

[119] E. Mendiluze Usandizaga, S. Ali, T. Yue, and P. Arcaini. Quantum circuit mutants: Empirical analysis and recommendations. *Empirical Software Engineering*, 30(3):100, 2025.

[120] Microsoft. Azure quantum development kit, 2024.

[121] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *Proceedings of the 1st International Conference on Learning Representations*, 2013.

[122] M. Monperrus. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[123] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162, 2014.

[124] S. Moro, P. Rita, and P. Cortez. Bank Marketing. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C5K306.

[125] J. M. Murillo, J. Garcia-Alonso, E. Moguel, J. Barzen, F. Leymann, S. Ali, T. Yue, P. Arcaini, R. Pérez-Castillo, I. García-Rodríguez de Guzmán, et al. Quantum software engineering: Roadmap and challenges ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–48, 2025.

[126] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[127] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.

[128] T. Nakagawa, S. Tokumoto, S. Tokui, and F. Ishikawa. An experience report on regression-free repair of deep neural network model. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, pages 778–782, 2023.

[129] M. Nauta, J. Trienes, S. Pathak, E. Nguyen, M. Peters, Y. Schmitt, J. Schlötterer, M. van Keulen, and C. Seifert. From anecdotal evidence to quantitative evaluation methods: A systematic review on evaluating explainable ai. *ACM Comput. Surv.*, 55(13s):42, 2023.

[130] B. News. Crypto exchange bybit confirms $1.5bn theft in one of the biggest digital heists. https://www.bbc.com/news/articles/c93dqpkwx4xo, February 2025. Accessed: 2025-11-05.

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

[132] M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[133] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[134] M. Nixon and A. Aguado. *Feature extraction and image processing for computer vision*. Academic press, 2019.

[135] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *Proceedings of the International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.

[136] OECD. AIM: AI Incidents and Hazards Monitor. https://oecd.ai/en/incidents, 2025. Accessed: 2025-09-02.

[137] OpenAI. ChatGPT. https://chat.openai.com/, 2022. Accessed: 2025-11-04.

[138] OpenAI. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/, 2025. Accessed: 2025-11-12.

[139] M. Openja, M. M. Morovati, L. An, F. Khomh, and M. Abidi. Technical debts and faults in open-source quantum software systems: An empirical study. *Journal of Systems and Software*, 193:111458, 2022.

[140] I. Ozkaya. What is really different in engineering ai-enabled systems? *IEEE Software*, 37(4):3–6, 2020.

[141] M. Paltenghi and M. Pradel. Bugs in quantum computing platforms: an empirical study. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

[142] B. Pang and L. Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 115–124, 2005.

[143] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[144] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[145] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering*, pages 609–620, 2017.

[146] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[147] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[148] M. Pezzè, S. Abrahão, B. Penzenstadler, D. Poshyvanyk, A. Roychoudhury, and T. Yue. A 2030 roadmap for software engineering. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–55, 2025.

[149] H. Qi, Z. Wang, Q. Guo, J. Chen, F. Juefei-Xu, F. Zhang, L. Ma, and J. Zhao. Archrepair: Block-level architecture-oriented repairing for deep neural networks. *ACM Transactions on Software Engineering and Methodology*, 32(5), 2023.

[150] Y. Qin, S. Wang, Y. Lou, J. Dong, K. Wang, X. Li, and X. Mao. Soapfl: A standard operating procedure for llm-based method-level fault localization. *IEEE Transactions on Software Engineering*, 51(4):1173–1187, 2025.

[151] M. O. Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.

[152] N. C. L. Ramalho, H. A. de Souza, and M. L. Chaim. Testing and debugging quantum programs: The road to 2030. *arXiv preprint arXiv:2405.09178*, 2024.

[153] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data:should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys. In *Proceedings of the Annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[154] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[155] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *Proceedings of 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013.

[156] H. Sak, A. Senior, and F. Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128*, 2014.

[157] S. Samsi, D. Zhao, J. McDonald, B. Li, A. Michaleas, M. Jones, W. Bergeron, J. Kepner, D. Tiwari, and V. Gadepally. From words to watts: Benchmarking the energy costs of large language model inference. In *Proceedings of the High Performance Extreme Computing Conference*, pages 1–9, 2023.

[158] N. Sato and R. Katsube. Locating buggy segments in quantum program debugging. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 26–31, 2024.

[159] N. Sato and R. Katsube. Bug-locating method based on statistical testing for quantum programs. *IEEE Transactions on Software Engineering*, 51(10):2804–2829, 2025.

[160] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28, 2015.

[161] C. E. Shannon and W. Weaver. *The mathematical theory of communication.* Univ. of Illinois Press, 1949.

[162] R. Shaydulin, C. Thomas, and P. Rodeghero. Making quantum computing open: Lessons from open source projects. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 451–455, 2020.

[163] D. Shen, G. Wu, and H.-I. Suk. Deep learning in medical image analysis. *Annual review of biomedical engineering*, 19:221–248, 2017.

[164] D. Sobania, A. Geiger, J. Callan, A. Brownlee, C. Hanna, R. Moussa, M. Z. López, J. Petke, and F. Sarro. Evaluating explanations for software patches generated by large language

models. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 147–152. Springer, 2023.

[165] J. Sohn, S. Kang, and S. Yoo. Arachne: Search-based repair of deep neural networks. *ACM Transactions on Software Engineering and Methodology*, 32(4), 2023.

[166] X. Song, Y. Sun, M. A. Mustafa, and L. Cordeiro. Airepair: A repair platform for neural networks. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings*, pages 98–101, 2023.

[167] D. Spinellis. *Effective debugging: 66 specific ways to debug software and systems*. Addison-Wesley Professional, 2016.

[168] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The german traffic sign recognition benchmark: a multi-class classification competition. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1453–1460, 2011.

[169] R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[170] S. Suganyadevi, V. Seethalakshmi, and K. Balasamy. A review on deep learning in medical image analysis. *International Journal of Multimedia Information Retrieval*, 11(1):19–38, 2022.

[171] B. Sun, J. Sun, L. H. Pham, and J. Shi. Causality-based neural network repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 338–349, 2022.

[172] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore. Testing deep neural networks. *arXiv preprint arXiv:1803.04792*, 2018.

[173] S. Tan, L. Lu, D. Xiang, T. Chu, C. Lang, J. Chen, X. Hu, and J. Yin. Hornbro: Homotopy-like method for automated quantum program repair. *Proceedings of the ACM on Software Engineering*, 2(FSE):734–756, 2025.

[174] Z. Tao, S. Nawas, J. Mitchell, and A. V. Thakur. Architecture-preserving provable repair of deep neural networks. In *Proceedings of the ACM on Programming Languages*, volume 7, pages 443–467. ACM New York, NY, USA, 2023.

[175] Y. Tian, K. Pei, S. Jana, and B. Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, pages 303–314, 2018.

[176] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi. Malware detection with deep neural network using process behavior. In *Proceedings of the 2016 IEEE 40th annual computer software and applications conference (COMPSAC)*, volume 2, pages 577–582, 2016.

[177] S. Tokui, S. Tokumoto, A. Yoshii, F. Ishikawa, T. Nakagawa, K. Munakata, and S. Kikuchi. Neurecover: Regression-controlled repair of deep neural networks with training history. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 1111–1121, 2022.

[178] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[179] X. Wang, T. Yu, P. Arcaini, T. Yue, and S. Ali. Mutation-based test generation for quantum programs with multi-objective search. In *Proceedings of the genetic and evolutionary computation conference*, pages 1345–1353, 2022.

[180] Z. Wang, Y. Huang, D. Song, L. Ma, and T. Zhang. Deepseer: Interactive rnn explanation and debugging via state abstraction. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2023.

[181] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang. Prioritizing test inputs for deep neural networks via mutation analysis. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*, pages 397–409, 2021.

[182] M. Wardat, B. D. Cruz, W. Le, and H. Rajan. Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *Proceedings of the IEEE/ACM 44the International Conference on Software Engineering*, pages 561–572, 2022.

[183] M. Wardat, W. Le, and H. Rajan. Deeplocalize: fault localization for deep neural networks. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*, pages 251–262, 2021.

[184] G. Weiss, Y. Goldberg, and E. Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *Proceedings of the 35th International Conference on Machine Learning*, pages 5247–5256, 2018.

[185] M. Weiss and P. Tonella. Simple techniques work surprisingly well for neural network test prioritization and active learning. In *Proceedings of the 31th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[186] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S.-C. Cheung. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering*, 47(11):2348–2368, 2019.

[187] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.

[188] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2011.

[189] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[190] W. E. Wong and Y. Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009.

[191] C. S. Xia, Y. Wei, and L. Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering*, pages 1482–1494. IEEE, 2023.

[192] C. S. Xia and L. Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 819–831, 2024.

[193] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[194] X. Xie, W. Guo, L. Ma, W. Le, J. Wang, L. Zhou, Y. Liu, and X. Xing. Rnnrepair: Automatic rnn repair via model-based analysis. In *Proceedings of the International Conference on Machine Learning*, pages 11383–11392, 2021.

[195] C. Xu, Z. Liu, X. Ren, G. Zhang, M. Liang, and D. Lo. Flexfl: Flexible and effective fault localization with open-source large language models. *IEEE Transactions on Software Engineering*, 51(5):1455–1471, 2025.

[196] X. Xu, V. Debroy, W. Eric Wong, and D. Guo. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering*, 21(06):803–827, 2011.

[197] S. Yan, Y. Xiong, K. Kundu, S. Yang, S. Deng, M. Wang, W. Xia, and S. Soatto. Positive-congruent training: Towards regression-free model updates. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14299–14308, 2021.

[198] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[199] Y. Yang, C. Li, Z. Han, R. Li, K. Xu, Q. Li, W. Zhong, Z. Shen, Z. Fei, J. Ge, and B. Luo. Patch generation in apr: A survey from the perspectives of utilizing llms and using apr-specific information. *ACM Transactions on Software Engineering and Methodology*, 2025. Just Accepted.

[200] B. Yu, H. Qi, Q. Guo, F. Juefei-Xu, X. Xie, L. Ma, and J. Zhao. Deeprepair: Style-guided repairing for deep neural networks in the real-world operational environment. *IEEE Transactions on Reliability*, 71(4):1401–1416, 2022.

[201] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 371–372, 2014.

[202] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, 8:58443–58469, 2020.

[203] J. Zappin, T. Stalnaker, O. Chaparro, and D. Poshyvanyk. When quantum meets classical: Characterizing hybrid quantum-classical issues discussed in developer forums. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pages 2931–2943, 2025.

[204] A. Zeller. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2009.

[205] H. Zhang and W. Chan. Apricot: a weight-adaptation approach to fixing deep learning models. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 376–387, 2019.

[206] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 48(1):1–36, 2022.

[207] P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and T. Dai. White-box fairness testing through adversarial sampling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 949–960, 2020.

[208] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–69, 2023.

[209] W. Zhang, Z. Li, Q. Wang, and J. Li. Finelocator: A novel approach to method-level fine-grained bug localization by query expansion. *Information and Software Technology*, 110:121–135, 2019.

[210] X. Zhang, X. Du, X. Xie, L. Ma, Y. Liu, and M. Sun. Decision-guided weighted automata extraction from recurrent neural networks. In *Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI)*, pages 11699–11707, 2021.

[211] X. Zhang, X. Xie, L. Ma, X. Du, Q. Hu, Y. Liu, J. Zhao, and M. Sun. Towards characterizing adversarial defects of deep learning software from the lens of uncertainty. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 739–751, 2020.

[212] X. Zhang, J. Zhai, S. Ma, and C. Shen. Autotrainer: An automatic dnn training problem detection and repair system. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*, pages 359–371, 2021.

[213] J. Zhao. Quantum software engineering: Landscapes and horizons. *arXiv preprint arXiv:2007.07047*, 2020.

[214] P. Zhao, Z. Miao, S. Lan, and J. Zhao. Bugs4q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs. *Journal of Systems and Software*, 205:111805, 2023.

[215] P. Zhao, X. Wu, J. Luo, Z. Li, and J. Zhao. An empirical study of bugs in quantum machine learning frameworks. In *2023 IEEE International Conference on Quantum Software (QSW)*, pages 68–75, 2023.

[216] P. Zhao, J. Zhao, and L. Ma. Identifying bug patterns in quantum programs. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pages 16–21, 2021.

[217] S. Zheng, Y. Song, T. Leung, and I. Goodfellow. Improving the robustness of deep neural networks via stability training. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4480–4488, 2016.

[218] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of 34th International conference on software engineering (ICSE)*, pages 14–24. IEEE, 2012.