The Role of Software Testing and Machine Learning in Automated Program Repair

PhD Thesis

Viktor Csuvik

Supervisor:
Dr. László Vidács

Doctoral School of Computer Science
Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged
2024

Contents

List of Figures

List of Tables

Glossary

AI Artificial Intelligence 8, 11, 18, 57, 58, 82, 112

APR Automated Program Repair 7, 8, 11, 14, 15, 16, 43, 44, 45, 60, 66, 71, 73, 74, 75, 80, 82, 83, 85, 86, 87, 89, 90, 91, 92, 99, 101, 105, 107, 108, 111, 112, 113

AST Abstract Syntax Tree 1, 14, 18, 26, 27, 28, 39, 42, 47, 58, 59, 66, 69, 93

CNN Convolutional Neural Network 48, 50, 55

DL Deep Learning 1, 8, 9, 12, 44, 49, 50, 54, 82, 112

DLFL Deep Learning Fault Localization 3, 48, 53

DOM Document Object Model 76

FL Fault Localization 7, 8, 9, 14, 44, 45, 81, 82, 83, 111, 112

G&V Generate-and-Validate 43, 44, 45, 66, 80, 85, 112

LLM Large Language Model 45, 81, 83

LSI Latent Semantic Indexing 18, 20, 21, 23, 24, 32, 37, 39, 42

ML Machine Learning 2, 7, 8, 9, 11, 12, 14, 17, 18, 21, 22, 26, 29, 43, 44, 49, 82, 86, 89, 93, 94, 95, 105, 106, 107, 112, 113

MLP Multi-Layer Perceptron 48, 50, 55, 86, 104, 106, 108

nDCG Normalized Discounted Cumulative Gain 91, 98, 99

NLP Natural Language Processing 14, 18, 27

OOP Object Oriented Programming 66

PCC Patch Correctness Check 8, 9, 85, 86, 89, 93, 94, 96, 103, 106, 107, 108, 109, 113

RNN Recurrent Neural Network 48, 50, 55

SBFL Spectrum-Based Fault Localization 1, 50, 52, 82, 83

SE Software Engineering 8, 49, 82, 93, 107, 112

TF-IDF Term Frequency-Inverse Document Frequency 9, 20, 21, 22, 23, 24, 32, 37, 38, 42, 111, 115

I INTRODUCTION

There is no perfect software. Every program can contain bugs, and most commercial system contains some kind of anomaly [95]. The basic approach to improve quality is software testing. This can provide an objective, independent view of the software, allowing businesses to assess and understand the risks of implementing software. Although testing can confirm the correctness of software by assuming certain specific hypotheses, it has limited ability to understand bugs and cannot improve the tested program. Testing cannot ensure that the product will work properly in all circumstances, but that is not the goal: rather, it is only to find circumstances in which this is not the case. In order to maintain quality, defects also need to be fixed, which is both time-consuming and resource intensive. In general, most of the budget for software projects will be spent on software maintenance and debugging [97].

This work dives into the several roles of software testing, traveling around several domains. First, a traceability problem is discussed: given test- and code classes, how to find a link between them. Without knowing the purpose of a test, its maintenance becomes cumbersome and it is of little significance. Next, Machine Learning (ML) approaches are applied on test-related tasks: Fault Localization (FL) and Automated Program Repair (APR). The role of test cases are multi-layered here: on one hand locating the bug usually relies heavily on test execution traces, on the other hand, fixes are generated by generating code that passes previously failed tests. Lastly, these automatically generated patches are supervised, as the test suite is usually not sufficient for *correct* patch generation (algorithms happen to overfit, thus generating patches for a program that only pass the test oracle, but are actually incorrect). These three main parts are connected by (1) the applied ML approaches, (2) presence of testing / test cases and (3) software quality improvement.

The theses is divided into three main chapters, each aligning with one of the three main points. The methodologies, experiments, and findings presented in the thesis have been extensively discussed in several of the author's prior works, nine of which are referenced here. Their relevance to the specific thesis points of the thesis is presented in Table 1.

Table 1: *Correspondence between the thesis points and publications.*

	Publications								
No.	[32]	[33]	[86]	[35]	[28]	[34]	[31]	[29]	[30]
I.	•	•	•						
II.				•	•	•			
III.							•	•	•

The thesis is organized as follows: the Background chapter, which follows, provides a concise overview of key concepts that traverse various chapters of this thesis.

The first part, Chapter 1 delves into textual methodologies aimed at identifying classes suitable for unit testing. It offers an overview of the significance and the latest advancements in test-to-code traceability techniques, indicating that the majority incorporate textual methods in their approaches. The chapter aims to conduct a thorough examination of the most prevalent textual techniques and explore potential avenues for their improvement.

The second part, Chapter 2 describes some ML applications in Software Engineering (SE). First, Fault Localization is assisted with Deep Learning (DL) techniques to enhance bug fixing, as the primary goal of testing is usually to identify software bugs so that they can be detected and corrected. Next, faulty programs are repaired automatically, using both modern and standard techniques. The chapter's main contributions include a method for stable Artificial Intelligence (AI) training, a dataset for learning-based APR training and tools that generate patches automatically.

The third part, Chapter 3 investigates possible solutions for the Patch Correctness Check (PCC) problem: that is, given an automatically generated fix for a program, decide whether it is really correct, or it overfits on the oracle. A similarity-based approach is proposed, as well a classification method that employs latest techniques in the field. Through profound investigations, the findings of the chapter indicate that improvement on overfitted patch detection can be achieved on certain level, thus improving the overall developer experience of Automated Program Repair (APR).

At the end of the thesis, brief summaries of the work are shown in English and in Hungarian, respectively. These, furthermore, contain brief summaries on the thesis points, as well as the author's contributions and publications.

Contributions

The ideas, figures, tables and results included in this thesis were published in scientific papers (listed at the end of the thesis). In a nutshell, the author is responsible

for the following contributions:

Chapter 1.: The author implemented the Doc2vec and TF-IDF methods for recovering traceability links. Additionally, he implemented the text-based recovery technique that retrieved call graph information from static code. The definition of the used source code representations and metric visualizations was also part of the author's work. He also took part in the evaluation and explanation of various other results, as well as in the planning and writing of all the published papers.

Chapter 2.: The author coordinated the experimentations on diverse network architectures on DL-based FL and implemented the bucketing approach. He also adapted churn metric and took part in the design and writing of the published paper. The FixJS benchmark creation and ChatGPT experiments were entirely the work of the author. In the GenProgJS tool, the author implemented the base genetic algorithm, and the interface for test case evaluation and operator calls. He also executed the experiments, coordinated the analysis and took a big part in the explanation of results.

Chapter 3.: The author laid the groundwork for the similarity-based PCC technique and implemented the base algorithm. He took part in the manual annotation of the generated patches. The author coordinated the implementation of the ML-based classifiers and conducted benchmark creation / gathering of all required metrics. He also planned the experiment guidelines and took a big role in the evaluation and explanation of the results and their implications.

Publications not included in the dissertation

[1] Márk Lajkó, **Viktor Csuvik**, Tibor Gyimóthy, and László Vidács Automated Program Repair with the GPT Family, including GPT-2, GPT-3 and CodeX. In *IEEE/ACM International Workshop on Automated Program Repair (APR)*, IEEE, 31-38, 2024.

[2] Dániel Horváth, **Viktor Csuvik**, Tibor Gyimóthy, and László Vidács An Extensive Study on Model Architecture and Program Representation in the Domain of Learning-based Automated Program Repair. In *IEEE/ACM International Workshop on Automated Program Repair (APR)*, IEEE, 31-38, 2023.

[3] Márk Lajkó, Dániel Horváth, **Viktor Csuvik** and László Vidács Fine-Tuning GPT-2 to Patch Programs, Is It Worth It?. In *Computational Science and Its Applications - ICCSA*, Springer, 79-91, 2022.

[4] Márk Lajkó, **Viktor Csuvik** and László Vidács Towards JavaScript program repair with Generative Pre-trained Transformer (GPT-2). In *IEEE/ACM International Workshop on Automated Program Repair (APR)*, IEEE, 61-68, 2022.

[5] András Kicsi, **Viktor Csuvik**, László Vidács, Ferenc Horváth, Árpád Beszédes, Tibor Gyimóthy and Ferenc Kocsis Feature analysis using information retrieval, community detection and structural analysis methods in product line adoption. *Journal of Systems and Software*, Volume(155), 70-90, 2019.

[6] András Kicsi, **Viktor Csuvik**, László Vidács, Árpád Beszédes and Tibor Gyimóthy Feature Level Complexity and Coupling Analysis in 4GL Systems. In *Computational Science and Its Applications - ICCSA*, Springer, 438-453, 2018.

[7] András Kicsi, László Vidács, **Viktor Csuvik**, Ferenc Horváth, Árpád Beszédes and Ferenc Kocsis Supporting Product Line Adoption by Combining Syntactic and Textual Feature Extraction. In *New Opportunities for Software Reuse - 17th International Conference, ICSR*, Springer, 148-163, 2018.

BACKGROUND

In the following chapters of the thesis I am going to talk about seemingly three distinct topics, but all of these are actually part of Automated Program Repair. As the task is vast and incredibly complex, subfields also tend to be inexhaustible and may operate on different domains of software engineering. In this chapter my goal is to establish basic knowledge that play crucial role in the following chapters - as some of the underlying concepts are of course common in all of the thesis points and some prior knowledge is necessary to understand the bigger picture.

Artificial Intelligence

To discover the link between human thinking and mechanical computers was already a concern of philosophers and mathematicians in the 17th and 18th centuries. However, questions related to Artificial Intelligence (AI) only emerged around the 1940s, but as a scientific field it became stable in 1956. Despite being a relatively young science, the literature on AI is very rich and expanding rapidly. Although hard to grasp the main goal of AI, a good way to think of it is to describe a hardware or software that allows a machine to mimic human intelligence. In this definition a lot of questions remain unanswered (e.g. what is intelligence in general), thus making AI an umbrella term covering many scientific fields and industries.

Machine learning

AI defines the goal to mimic human intelligence, but does not say anything how to do that. Machine Learning (ML) on the other hand, develops and applies algorithms that can learn from experience and improve over time without being explicitly programmed. Thus, it is a technique to imitate human intelligence. It collects large amounts of data, analyses and learns from it, then makes intelligent decisions. The goal of ML is to create programs that can improve their own efficiency by using the experience they gain during operation. Compared to a traditional program, where the programmer defines the behaviour, telling the machine how to behave and what to output in response to a given input is, in ML the ”programmer” only defines what the ”program” should be able to do and given the input, tells it to learn the output

- so in most cases it has no control over it (there are exceptions, of course). In case of ML, usually the machine defines (learns) the rules based on the input, whereas in the case of classical software, it is the programmers who do this.

Neural Networks

Incredible as it may seem, the basis of neural networks that now seems almost magical was laid a long time ago, in 1957, with the so-called perceptron model. The idea was to mimic human intelligence by using mathematics to simulate the functions of nerve cells (neurons) in the human brain. The human brain is estimated to have 50-100 billion such neurons, and this initial approach mimicked the function of only 1 such neuron. Its practical use is very limited, and research in this area has been neglected in the following decades. The perceptron model was later developed into multilayer neural networks. Like most good ideas in general, this one is also very natural: connect neurons in a similar way to how they are connected in the human brain, thus forming a network. Neurons are connected to each other, usually they are organized in layers. This neural network needs to be trained to complete certain tasks, during the learning process the network adjusts the weights of the connections to achieve the best possible result. As more examples are available and computation power became more accessible, deep neural networks emerged. They have more hidden layers than traditional neural networks to solve more complex tasks. The advantage of them is that they can learn more complex correlations as they process data in more layers. It can also be shown that as the network goes deeper, it manages to recognize more abstract concepts. The disadvantage of Deep Learning (DL) is that they are more difficult to train (technical problems can arise), and more data is required. Nevertheless, most approaches today work with deep neural networks.

Transformers

Traditional neural networks are also called as Feedforward Neural Networks (FNN) because data always flows ”forward” in them. In theory every practical problem can be solved using these, but the practice shows that in special tasks, special network architectures are more advantageous. For example it is a common practice to use Convolutional Neural Networks (CNNs) for image-related tasks, while Recurrent Neural Netowkrs (RNNs) in tasks where the order of data is important. The Transformer model was introduced recently and overperformed previous architectures in a lot of tasks [170]. The innovative approach of this model changes the way traditional neural networks are built and operate in a way that makes it more efficient to handle sequential data. It has no classical layers and consists of two main components: an encoder and a decoder. The structure of the two components is similar, but the data between them does not flow sequentially from one layer to the other. Instead, the

so-called attention mechanism plays a very important role. This allows the model to weight and consider different parts of the input sequence while processing the data. It is very useful, for example, when processing longer texts, as the model can decide on which part of the text pay more attention. The attention mechanism determines the importance of the relationships between words.

Document embeddings - Doc2Vec

Doc2Vec is originated from Word2Vec, which was introduced by Google's developers in [123]. Word2Vec encodes words into vectors containing real numbers with a neural network, these are called word embeddings. The basic idea is the following: for a given surrounding, the model predicts the current word (CBOW model) or the prediction goes in the opposite direction (Skip-gram model). The trick is that the hidden layer of the shallow neural network used has fewer neurons than the input and output layers, forcing the model to learn a compact representation. The weights in the hidden layers will provide the word embeddings and the number of neurons will be the dimension of the embedding. Doc2Vec differs only in small details: it can encode whole documents by adding a unique identifier of the document to the input layer. This way a word can have multiple embeddings in different documents (which is more realistic in some cases, e.g.: blue, bear). Utilizing the embeddings, one can compute the similarity between documents.

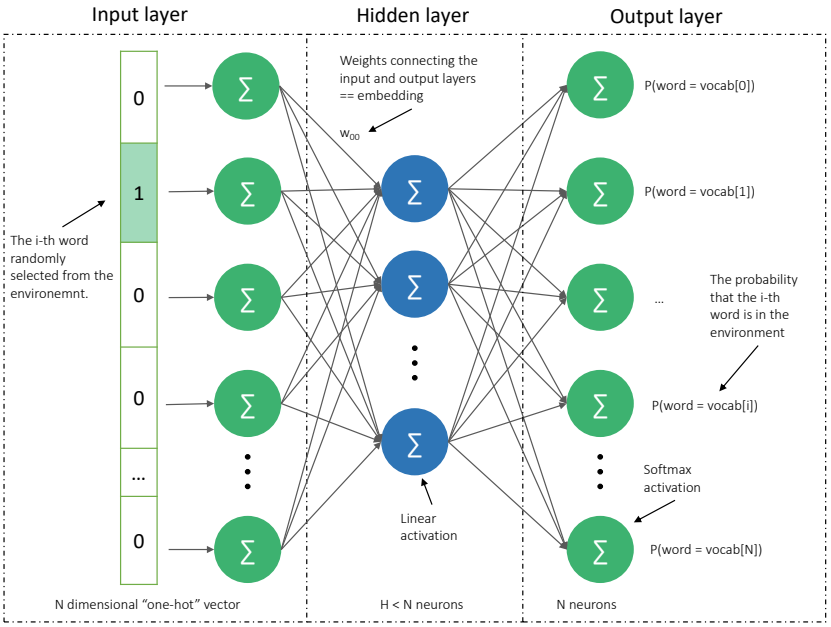


Figure 1: *General architecture of the embedding model.*

Software Testing

Testing constitutes a major aspect in the assurance of the quality of a software. Besides simply indicating faults in software, tests are also essential for other areas in software engineering, like code maintenance, Fault Localization or Automated Program Repair. The primary aspect of testing is to provide information on whether the software achieves the general result its stakeholders desire. Testing can provide an independent view of the software and opens new opportunities in calculating the risks. It is known that complete testing is not fully achievable, still writing tests on edge cases and increasing their amount is considered to be a good coding practice. It is not a coincidence that large systems often incorporate vast amounts of tests.

Source Code Representation

From a technical point of view, source code is textual information, thus it can serve as input to any Machine Learning model that requires such data format. However, it is not straightforward how the source code need to be represented to such models, as the structure of the source code holds some underlying information. The Abstract Syntax Tree (AST) of a source code is a tree representation of the abstract syntactic structure of that text. Parsing ASTs into a feasible input format is not a straightforward process as different tasks might require different aspects from it. In Natural Language Processing (NLP), words and sentences are usually split into tokens. Tokens are the frequent character sequences in the text, i.e. the word fragments. The process that generates tokens from running text is called tokenization, which may vary from approach to approach, but the essence is the same: generating tokens that are the building blocks of words (including lemmatization, stemming, etc.). In case of source code the word boundries by which the splitting is done is less intuitive: special characters, keywords and not meaningful words also occur in the text.

Automated Program Repair

Automated Program Repair (APR) is a field of software engineering that aims to automatically fix defects in computer programs. APR has the potential to significantly improve software reliability and reduce the cost and time associated with manual debugging and repair [174]. There are several near equivalents of software failure, it is important to distinguish between them so that we can be precise in the following:

- Error/Mistake: human error leading to incorrect results.
- Fault/Defect: is the manifestation of the *Error* in the code, i.e. when the error occurs in the program (we call this *bug*).

- Failure: a deviation from the expected functionality of the software caused by a bug.

It is the latter that can typically cause great damage to companies. Therefore it is best to avoid these mistakes, so that maintenance costs do not skyrocket. One solution to this is Automated Program Repair, which is designed to fix defects before they become failures. By definition, the process by which defects in software are automatically fixed [126]. This is a rather general formulation, thus several approaches have been introduced to tackle with the problem. However, the goal is well stated: given a program, as many bugs as possible should be corrected *automatically* - without human intervention. It is a rather complex task, often even finding the location of the error itself can be rather difficult. Several Automated Program Repair approaches have been introduced, their skeleton is similar [11], and they mainly consist of the steps depicted on Figure 2.

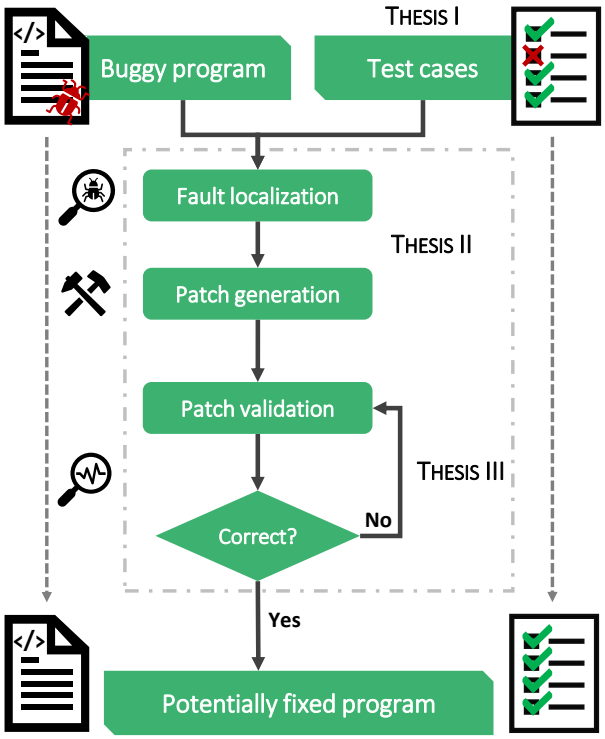


Figure 2: *General approach of Automated Program Repair*

1. Fault localization: determining the source of the error, finding the buggy module (this part can also include the detection and diagnosis of the anomaly). The output is usually a list of suspicious statements that needs to be repaired, but more recently the assumption of perfect fault localization is also widespread among works focusing only on patch synthesis.

2. Patch generation: taking steps to eliminate the misbehaviour. Typically, there are more than one correct fix for a bug and these should be tested or scored. The program under repair is usually referred to as a *program variant* in the literature, while the set of changes is the *patch* [11, 21, 70]. The ”goodness” of variants is evaluated using a so-called *fitness function*.

3. Patch validation: validate the correctness of a variant. If a previously existing error has been fixed, the algorithm stops, otherwise it starts to make a new variant.

The output of the process is a potentially fixed program. The reason of having only a *potentially* fixed program is that despite validating the patch, it is usually not correct (thus the validation step is not sufficiently thorough). The reason of *potentially* is that it is usually necessary to have experienced software developers check the patch - the automatically generated program, despite having behaved correctly in testing, is often not correct. As can be seen on Figure 2 it is the input test cases that reveal the incorrect behaviour (by having at least 1 failing test case), and most often they play an important role in the repair process as well. Tests are also often used to validate patches: if the patched program passes all test cases, the patch is considered to be potentially correct. In fact, patch generation is also often based on tests [159, 192]. Thus, the simplest way to capture the basic task of APR is: given a buggy program and a test database *with at least one failing test*, generate a patch for it, which causes all test cases to pass [177]. It is clear that automated program repair is closely linked to tests, their existence is crucial for a correct repair process.

The conventional APR approach is to generate a *patch* (e.g., using genetic algorithm) and then validate it against an *oracle* (i.e., test suite). Although these approaches have been criticized several times, they still define the research direction of APR [83]. Their standalone and easy-to-use nature makes them competitive against learning-based approaches [107]. On the other hand, data-driven APR approaches utilize machine learning techniques to learn from a dataset of programs and their corresponding repair patches. These approaches often require a huge train-test-validate dataset to adapt to different repair strategies and programming languages [113, 196]. The training of such methods is resource-intensive, and the approaches are often not usable due to availability issues (e.g., confidentiality), executability concerns (e.g., specific execution environment), or configurability limitations [83].

1 Textual Similarity Techniques in Code Level Traceability

1.1 Overview

Test-to-code traceability means finding the links between test cases and production code. More precisely for a test case we want to find certain parts of the code which it was meant to test. For a large system, this task can be challenging, particularly when the development lacks good coding practices [180] like proper naming conventions. Using practices like naming the test classes after the tested production code automatically creates a conceivable link between the test and the tested artifact. It is well known that with proper naming conventions, retrieving traceability links is a minor task [152]. If we consider, however, a system where the targets of the test cases are unknown to us, other approaches should be applied.

Considering tens of thousands of tests in a software system, their maintenance becomes cumbersome and the goal of some tests may even become unknown. In these cases recovering which test case assesses a specific part of code can prove to be a challenge. Traceability in general stands for the task of tracing software items through a variety of software products. The previously described specific problem is called *test-to-code* traceability. Traceability is a well-researched area with a serious industrial background. While the most widespread problem in this field is domain requirement traceability [9, 115], test-to-code traceability also gained attention from the research community [33, 152].

Using good coding practices [180] can make the task easier and with proper naming conventions [152] very accurate results can be achieved. However, if a developer lacks these skills or proper foresight, the traceability problem becomes non-trivial. In these cases, automatic recovery approaches should be introduced, which does not require such assumptions from the examined system. While several attempts have already been made to cope with this problem, these techniques are limited since they typically depend on intuitive features. In this chapter a method is described, that automatically links test cases and production classes relying only on conceptual information and an attempt to improve results by involving new ML techniques.

In its most basic form, adhering to naming conventions entails that the name of a test case should correspond to the name of the production code element it is designed to test. Specifically, the name of the test case should include the name of the target class or method along with the term ”test”. Moreover, the test should maintain the same package hierarchy as the production code it targets. According to a 2009 study by Rompaey and Demeyer [152], the application of naming conventions during development can achieve complete precision in detecting traceability links. However, enforcing these conventions is challenging and heavily reliant on developer practices. Additionally, method-level naming conventions present various complicating factors.

Certain recovery techniques utilize structural or semantic information within the code that is less dependent on individual working practices. One such technique is based on information retrieval (IR), which primarily leverages textual information extracted from the system's source code. In addition to the source code, other forms of non-textual information can be derived, such as the Abstract Syntax Tree (AST) or other structural descriptors. While source code syntax is highly formalized with predefined language keywords, it also typically contains a significant amount of unregulated natural text, including variable names and comments. The naming of variables, functions, and classes is highly variable and usually meaningful. Although source code is challenging for humans to interpret as natural language, Machine Learning (ML) methods commonly used in Natural Language Processing (NLP) can still be effectively applied.

Compared to a small manual dataset, Rompaey and Demeyer [152] found that lexical analysis (Latent Semantic Indexing (LSI)) applied to this task performed with 3.7%-13% precision while the other methods all achieved better results. Thus, it is known that IR-based methods most probably do not produce the best results in the test-to-code traceability field. However, these techniques are continually employed in current state-of-the-art solutions. While textual methods may not be the most effective means of producing valid traceability links, modern approaches still incorporate them alongside other techniques. The textual methods used in these systems are often outdated, with many solutions relying on simple class name matching or the Latent Semantic Indexing (LSI) technique as part of their contextual coupling. Consequently, identifying more effective textual methods can enhance these combinations, potentially making significant contributions to the field. Research findings indicate that improved lexical analysis methods can substantially outperform previous approaches, increasing average precision to over 50%. To recover test-to-code traceability links based solely on source files, an appropriate input representation must first be generated, followed by training an AI model to search for the most similar test-to-code matches.

1.2 Related Work

Traceability in software engineering research typically refers to the discovery of traceability links from requirements or related natural text documentation towards the source code [9, 115]. Based on the study of Borg et al. [16], most of the traceability evaluations have been conducted on small bipartite datasets containing fewer than 500 artifacts, which is a severe threat to external validity. While data limitations still persist, the current paper's evaluation is conducted on eight software systems, using different oracles. While to the best of our knowledge, test-to-code traceability is not the most widespread topic amongst other recovery tasks, several well-known approaches aim to cope with this problem. Still, as yet, none of them has provided a perfect solution for the problem [32, 33, 84, 152]. The current state-of-the-art techniques [147] rely on a combination of diverse methods - i.e. techniques based on dynamic slicing and contextual coupling. The use of textual information is common in these techniques. Our current work took a closer look at various textual similarity techniques, and combinations of these resulted in promising recovery precision.

In a recent work [179], authors presented TCtracer, a tool which combines an ensemble of new and existing techniques and exploits a synergistic flow of information between the method and class levels. The tool observes test executions and create candidate links between these artefacts and the tested artefacts. It then assigns scores (which are used to rank the candidates) to the candidate links. These scores are calculated using the combination of eight test-to-code traceability techniques including four string-based techniques, two statistical, call-based techniques, Last Call Before Assert (LCBA) and Naming Conventions (NC). Although this and our work share many common factors, there are significant differences. First of all, our technique does not rely on information based on test-execution. Secondly, the two rankings are fundamentally different: our work relies on IR techniques (and refine these using various approaches, with an initial static analysis), while White et al. calculate the ranking scores based on formulas defined in the paper. Finally, we also researched different ways of representing the source code.

The utilization of structural information has also occurred in other works [136, 147, 148]. In their 2015 work, Ghafari et al. [54] also employed structural information. Here, the main goal was to identify traceability links between test cases and methods under test, which is still not a mainstream topic in the field, as most methods aim for production classes. The proposed approach correctly detects focal methods under test automatically in 85% of the cases. Bouillon et al. leveraged a failed test case to find the location of errors in source code [17]. To link the tests to the production code, they built the static call graph of each test method and annotated each test with a list of methods which may be invoked from the test. The use of structural information also occurs in other extraction methods, feature extraction for

instance, where it was shown that its combination with LSI is capable of producing good results [46]. In the current paper, structural information was used in several source code representations. Call information was also utilized, even though it was extracted only from the text. Even so, it was found a valuable addition as a filter.

Like LSI, TF-IDF is also a text-based model commonly used in the software engineering domain. This technique was, for instance, used by Yalda et al. [191] to trace textual requirement elements to related textual defect reports. In requirement traceability, the use of TF-IDF is so widespread, that it is considered a baseline method [163]. Text-based models are still very popular in the requirement traceability task also, they are incorporated in several recent publications [50, 68]. Our experiments covered LSI and TF-IDF as standalone techniques and also as a refinement for Doc2Vec, which was shown in our previous work [32] to produce higher quality results.

In our findings, the use of document embeddings resulted in the highest precision values. Word2Vec [123] gained a lot of attention in recent years and became a very popular approach in natural language processing. Calculating similarity between text elements using word embeddings became a mainstream process [120]. Doc2Vec [123] is an extension of the Word2Vec method dealing with whole documents rather than single words. Although not enjoying the immense popularity of Word2Vec, its use is still prominent in the scientific community [39, 173, 203].

The use of recommendation systems is widespread in the field of software engineering [91]. Presenting a prioritized list of most likely solutions seems to be a more resilient approach even in traceability research [33, 84].

Because of the numerous benefits of tests, developers tend to create a lot of them even though it is challenging to determine what new tests to add to improve the quality of a test suite. Since 100% coverage is often infeasible, several new approaches have been proposed for interpreting coverage information. For instance, Huo et al. [74] introduced the concepts of direct coverage and indirect coverage, that address these limitations. In addition, several other challenges are present in general software testing [13], like coherent testing, test oracles and compositional testing. The more challenges are solved, and the more the community understands about testing in general, the better test-to-code traceability results can become [137]. The current paper also aimed to shed some light on class and method naming habits which can lead to a better understanding of testing in real-life software systems.

Although natural language based methods are not the most effective standalone techniques, state-of-the-art test-to-code traceability methods like the method provided by Qusef et al. [147, 148] incorporate textual analysis for more precise recovery. Jin et al. in [61] presented a solution that uses deep learning and word embeddings to incorporate requirements artifact semantics and domain knowledge into the tracing solution. The authors evaluated their approach against LSI and VSM

(Vector Space Model). They found that their neural-based approach only outperforms these when the tracing network has a large enough training data which is hard to obtain. Other works also explore the use of word embeddings to recover traceability links [61, 62]. Our current approach differs from these in many aspects. To begin with, we make use of different similarity concepts and further refine these with structural information. Next, our document embeddings are computed in one step, while in other approaches this is usually achieved in several steps. Finally, our models were trained only on source code (or on some representation which was obtained from it), and there was no additional natural language corpus.

1.3 Method

Figure 1.3 provides an illustration of the comprehensive proposed approach. A software system written in Java programming language is considered, which contains both test classes and production code and the aim is to recover the relationship between them. No assumptions were made about the names of the software artifacts. From the raw source code, classes, production code and test cases of the system are extracted using the Source Meter [1] static analysis tool. This also includes the extraction of *soft computed call graph information*, which essentially is an additional structural data that might help in the linking process. Five diverse representations of the source code is generated and we use ML techniques to measure the similarity between code snippets. In the case of Latent Semantic Indexing (LSI) and Term Frequency-Inverse Document Frequency (TF-IDF) methods the models are trained on the production code (corpus) and the test cases are the queries. There is a slight difference in the case of Doc2Vec since the training corpus consists of both the test and production classes. After the models are trained, we measure the similarity between tests and code classes, from which a ranked list is constructed. This ranked list is then fine-tuned using the previously extracted call graph information. The basic idea is that test and code classes are *similar* in some sense and that additional structural information will filter out the errors of the previous steps. Therefore, from the ranked similarity list, one can observe the first N production classes, allowing to consider these techniques as recommendation systems.

In this work classes are recommended for a test case starting from the most similar and also the top 2 and top 5 most similar classes are examined. Looking at the outputs in such a way holds a number of benefits. Foremost, if only the most similar class would be extracted then those instances when tests assess the proper functioning of several classes rather than only one would be missed. Also, a class usually relies on other classes, consequently a recommendation system can highlight

[1] https://www.sourcemeter.com/

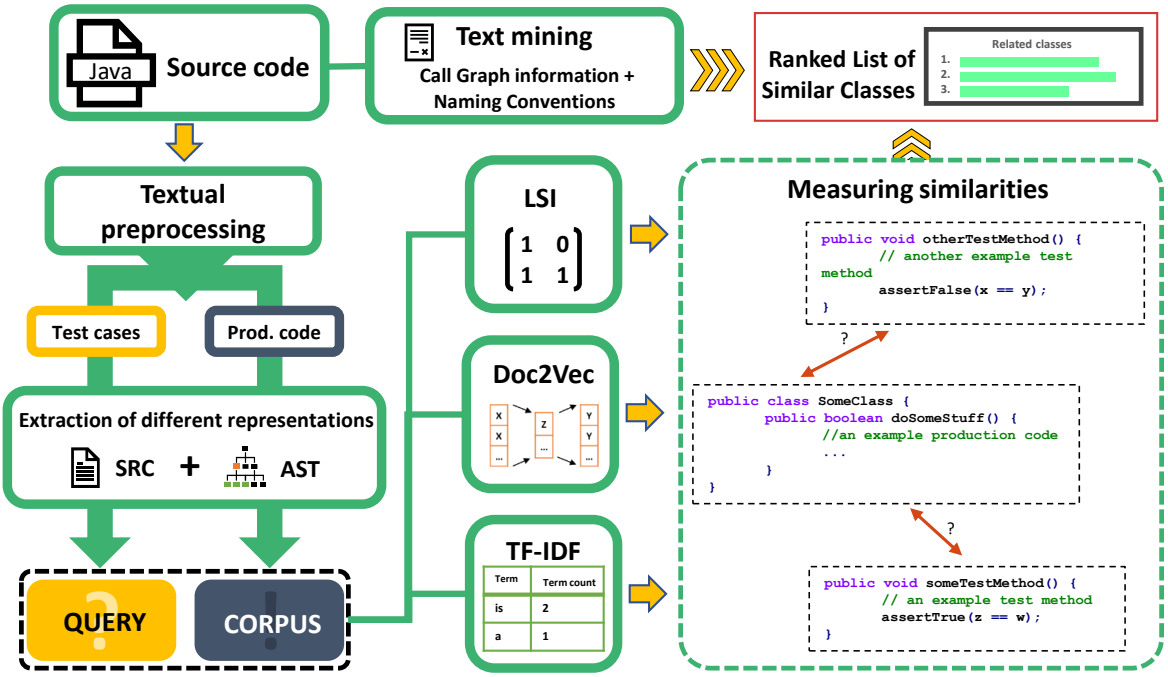


Figure 1.3: *A high-level illustration of linking test cases to code classes.*

the test and code relationship more thoroughly. Since overly abundant recommendations can result in a high number of false matches which can diminish the usefulness of the information itself, we restricted the consideration to only the 5 most similar classes in each case, keeping the technique as simple as possible. In the upcoming subsections, the used techniques are explained to obtain the similarity between two parts of the source code. The Gensim [1] toolkits implementation was used for all three ML methods.

1.3.1 Term Frequency–Inverse Document Frequency: TF-IDF

Term Frequency-Inverse Document Frequency (TF-IDF) is an information retrieval method, that relies on numerical statistics reflecting how important a word is to a document in a corpus [99]. It is basically a metric and its value increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. One can compute TF-IDF by multiplying a local component (term frequency) with a global component (inverse document frequency) and normalizing the resulting documents to unit length. The formula for a non-normalized weight of term i in document j in a corpus of D documents is displayed in Equation 1.1. One of the simplest ranking functions is computed by summing the weights for each query term, but many more sophisticated ranking functions also exist [69].

$$weight_{ij} = \left(frequency_{ij} * log_2 \frac{D}{DocumentFrequency_i} \right) \tag{1.1}$$

1.3.2 Document embeddings: Doc2Vec

Doc2Vec is a shallow neural network that can produce document embeddings. The detailed description of it can be found in Chapter . Here, the 3COSMUL metric, proposed in [102], displayed in Equation 1.2 was used, to measure similarity between the vectors. The choice of similarity metric is arbitrary and the selection of it was out of scope of the current research.

$$arg\max_{b* \in V} \left(\frac{cos(b*, b) cos(b*, a*)}{cos(b*, a) + \epsilon} \right) \tag{1.2}$$

1.3.3 Latent Semantic Indexing: LSI

Latent Semantic Indexing (LSI) is a technique in natural language processing of analyzing the relationships between documents. During the learning procedure, a matrix is constructed, which contains word counts. The elements inside the matrix are typically weighted with the TF-IDF values, but note that the base process differs from the previous one. The main idea of LSI is that the matrix is transformed into a lower dimension using singular value decomposition and in the resulting matrix the conceptually more similar elements get more similar representations. The most similar documents to a query can easily be found as the query also represents a multidimensional matrix with which a suitable distance method can rank each document by similarity.

1.3.4 Result Refinement With an *Ensemble* Technique

After test and code classes had been separated and the code representations had been obtained, three models separately have been trained and investigated the similarities. In Figure 1.4 the ranked lists of the three alternative methods trained are

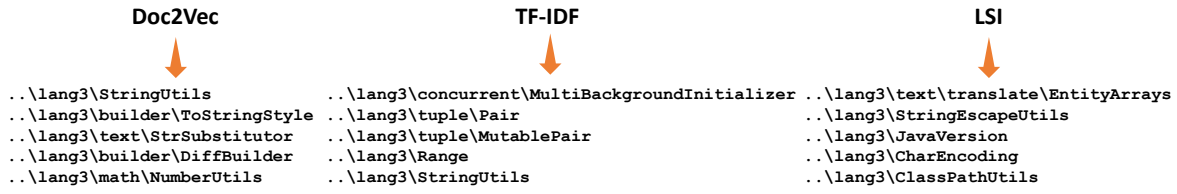


Figure 1.4: *Ranked lists produced by different approaches for the* StringUtilsSubstringTest *test class.*

shown. For a given test class (*StringUtilsSubstringTest*) only the Doc2Vec method classified the desired code class (*StringUtils*) as the most similar (while of course in a different case another one of the methods could provide the desired class). Additionally note, that TF-IDF put the *StringUtils* class in the fifth place, while LSI did not rank it among the top-5 most similar classes (it was in the eleventh place on the ranked list). This example demonstrates that the ranked list of each technique can contain useful information, the desired code class appears close to the top of every list. Thus it can be possible to refine the obtained results one technique provides with the list of other techniques. A simple algorithm has been defined to achieve this goal, which is shown in Listing 3.1. The ranked list obtained from the first method is filtered with the second methods ranked list. Since the ranked lists contain every code class, these are limited up to the top 100 most similar classes, this way the featured algorithm will drop out classes from the first if those are not present on the second ranked list. Note, that this refinement procedure cannot introduce new classes to the first ranked list, only removes them.

```
1  # ranked_list_i: ranked list from the i-th technique,
2  # which contains the top 100 most similar classes
3  result = []
4  for code_class in ranked_list_1:
5    if code_class in ranked_list_2:
6      result.append(code_class)
```

Listing 3.1: *Algorithm used to refine the obtained similarity lists.*

1.3.5 Soft computed call information

Since the listed techniques do not consider class information, I contributed an additional simple filter to enhance the accuracy. This filter is based on two primary assumptions: (1) the package of the class under test should either be the same as that of the test or it should be imported in the test, and (2) a valid target class should define at least one method that is invoked within the body of the test case. Although these criteria do not guarantee a valid match, they provide a more focused approach. To extract methods and imports from Java files, I utilized regular expressions tailored to the syntax of the Java programming language. Specifically, the regular expressions were designed to identify method definitions and import statements within the code. It is important to note that these regular expressions would need to be adapted for different programming languages due to variations in syntax. This method of using regular expressions to parse Java files allows for the precise extraction of relevant structural elements, thus enhancing the reliability of the traceability links.

Extended Naming Convention Extraction

The techniques described above yield a filtered list of soft computed links, without guaranteeing their correctness. Naming conventions, however, are known to produce traceability links with very high precision [152]. If a project lacks proper naming practices, these conventions cannot be used to identify correct matches. In the final approach, naming conventions are applied first. If applicable, they are accepted. Otherwise, results from an IR-based approach are considered.

1.4 Data Collection and Source Code Representations

The designed approach is applicable to projects written in Java, one of the most popular programming languages in use [2]. In general, the featured technique is independent of text representations, so the programming language of the source code is not necessarily important. In Table 1.2 one can observe the projects, on which the proposed technique is evaluated.

Table 1.2: *Size and versions of the programs used.*

Program	Url (github.com/)	Version	Classes	Methods	Test methods
ArgoUML	argouml-tigris-org/argouml	0.35.1	2 404	17 948	554
C. Lang	apache/commons-lang	3.4	596	6 523	2 473
C. Math	apache/commons-math	3.4.1	2 033	14 837	3 493
Gson	google/gson	2.8.0	757	2 467	924
JFreeChart	jfree/jfreechart	1.0.19	953	11 594	2 239
Joda-Time	JodaOrg/joda-time	2.9.6	522	9 934	3 779
Mondrian	pentaho/mondrian	3.0.4.11371	1 626	12 186	1 546
PMD	pmd/pmd	5.6.0	1 608	9 242	825

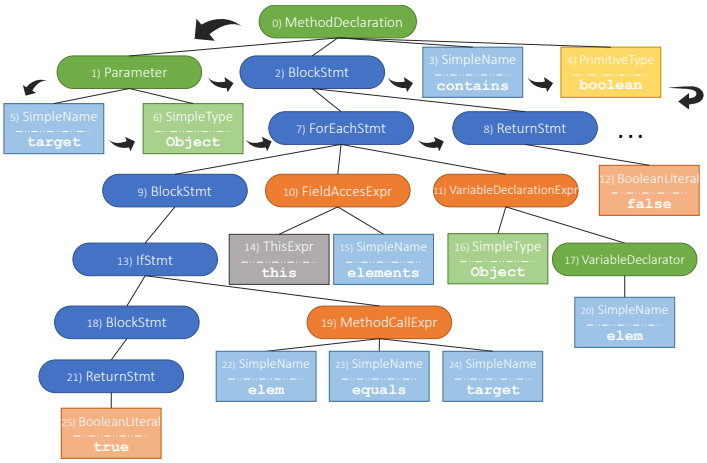
The choice of the projects was influenced by several factors: (1) the projects should be publicly available, so source code can be obtained (2) proper naming convention were followed to some extent. Two of the systems strive to have minimal dependencies on other libraries [10] and are modules of the Apache Commons project, these are Commons Lang and Commons Math. Lang aims to broaden the functionality provided by Java regarding the manipulation of Java classes, while Math provides mathematical and statistical functions missing from the Java language. Mondrian has a large development history (the development was started in 1997 [125]) and is an open source Online Analytical Processing system, which enables high-performance analysis on massive amounts of data. JFreeChart is a relatively new software, its first release was in 2013. The project is one of the most popular open source charting

tools. Gson is a Java library that does conversions between Java objects and Json format efficiently and comfortably. PMD is a tool for program code analysis. It explores frequent coding mistakes and supports multiple programming languages.

It is evident that the exact contents of the input are of crucial importance. In the following sections, the representations of code snippets (classes or methods) are described. A code representation is the input of a Machine Learning algorithm that computes the similarity between distinct items. Abstract Syntax Tree were utilized to form a sequence of tokens from the structured source code. An AST is a tree that represents the syntactic structure of the source code, without including all details like punctuation and delimiters. For instance, a sample Abstract Syntax Tree and source code sample is displayed in Figure 1.5. To better understand the advantages and best possible methods of using the AST, the paper describes experiments on five different code representations, of which four relies on AST information. The five chosen representations are described below. The five representations under evaluation were constructed according to the most widely used representations in other research experiments [178], constructed along the work of Tufano *et al.* [169]. In order to help understanding the difference between each representation, an example is shown in Figure 1.6.

```
boolean contains(Object target) {
    for (Object elem: this.elements) {
        if (elem.equals(target)) {
            return true;
        }
    }
    return false;
}
```

(a) *An example method declaration.*



(b) *An AST generated from the source code.*

Figure 1.5: *Source code and Abstract Syntax Tree. The numbers inside each element indicate the place of the node in the visiting order. Leaves are denoted with standard rectangles (note that here the value and the type is also represented), while intermediate nodes are represented by rectangles with rounded corners.*

SRC

Let us consider the source code as a structured text file. In this simple case, similar methods are used in the context of Natural Language Processing (NLP). These techniques include the tokenization of sentences into separate words and the application of stemming. With natural language, the separation of words can be quite simple. In the case of source code, however, we should consider other factors as well. For instance, compound words are usually written by the camel case rule, while class and method names can be separated by punctuation. The definition of these separators are one of the main design decisions in this representation. For the current work words were split by the camel case rule, by white spaces and by special characters that are specific to Java (”(”, ”[”, ”.”). The Porter stemming algorithm was used for stemming. This approach notably does not use the AST of the files, making it a truly only text-based approach.

AST

To extract this representation for a code fragment, an Abstract Syntax Tree (AST) has to be constructed. This process ignores comments, blank lines, punctuation, and delimiters. Each node of the AST represents an entity occurring in the source code. For instance, the root node of a class (CompilationUnit) represents the whole source file, while the leaves are the identifiers in the code. In this particular case, the types of AST nodes were used for the representation. The sequence of symbols was obtained by pre-order traversal of the AST. The extracted sequences have a limited number of symbols, providing a high-level representation.

IDENT

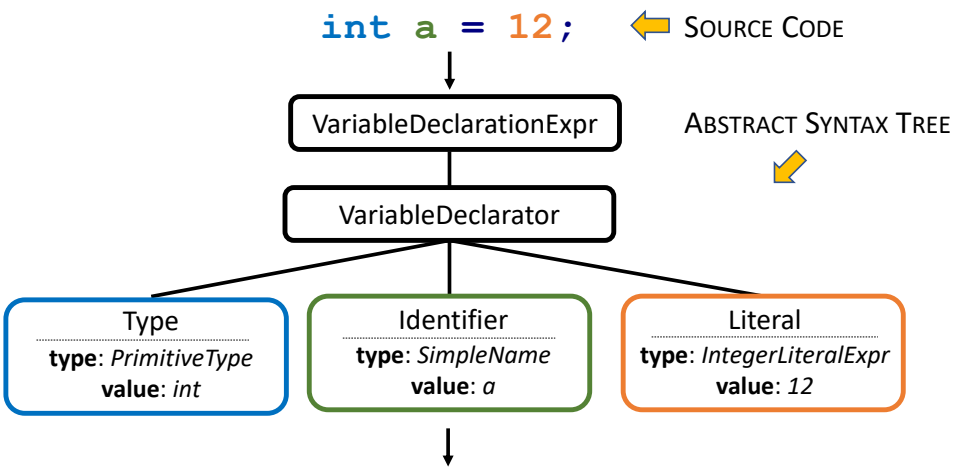
Every node in the Abstract Syntax Tree (AST) has a type and a value. The top nodes of the AST correspond to a higher level of abstraction (like statements or blocks), their values typically consist of several lines of code. The values of the leaf nodes are the keywords used in the code fragment. In this representation, these identifiers are used by traversing the AST tree and printing out the values of the leaves. The values of literals (constants) in the source code also might occur here, these are replaced with placeholders representing their type (e.g. an integer literal is replaced with the *¡INT¿* placeholder, while a string literal with *¡STRING¿*). The extracted identifiers contain variable names. In the current experiments, they were split according to the camel case rule popularly used in Java.

LEAF

In the previous two representations, distinct parts of the AST were utilized to get the input. This approach takes both the types and node values into account. Just as before, a pre-order visit is performed from the root. If the node is an inner node then its type, otherwise (when it is a leaf) its value is printed. This representation captures both the abstract structure of the AST and the code-specific identifiers. Considering the latter, these can be very unique and thus very specific to a class or a method.

SIMPLE

The extraction process is very similar to the previous one, except that in this case only values with a node type of *SimpleName* are printed out. These nodes occur very often, they constituted 46% of an AST on average in the experiments. These values correspond to the names of the variables used in the source code while other leaf node types like literal expressions or primitive types hold very specific information. Note that in the IDENT representation, the replacing of literals eliminated the AST node types of literal expressions. Only the modifiers, names, and types remained, thus becoming similar to this representation. With this representation, however, we do not exclude the inner structure of the Abstract Syntax Tree (AST).



(1) **SRC** - int a = 12 ;
(2) **AST** - VariableDeclarationExpr VariableDeclarator PrimitiveType SimpleName IntegerLiteralExpr
(3) **IDENT** - int a <INT>
(4) **LEAF** - VariableDeclarationExpr VariableDeclarator int a <INT>
(5) **SIMPLE** - VariableDeclarationExpr VariableDeclarator VariableDeclarator a IntegerLiteralExpr

Figure 1.6: *Example source code and extracted representations.*

1.5 Evaluation Procedure

To evaluate the text-based methods, one should know whether the proposed ML techniques recommend the correct production class for a given test case. To achieve this, we used existing naming conventions, which are based on package hierarchy and exact name matching (see Section 1.5 for further details). Due to the potential influence of naming convention habits, we also evaluated the approach using a human test oracle as described in [85]. TestRoutes is a manually curated dataset containing data on four of our eight subject systems: Commons Lang, Gson, JFreeChart, and Joda-Time. This dataset is suitable for class-level evaluations, a relaxed version of the problem. It lists the methods under test as focal methods (with multiple focal methods possible for a test case), as well as test and production context. Our current focus is on the classes associated with these focal methods. For JFreeChart and Joda-Time, the dataset specifically targeted test cases not covered by simple naming conventions, as reflected in the results. For the other systems, the dataset includes data on randomly selected test cases. The TestRoutes data was annotated by a graduate student familiar with software testing. During the annotation process, the tests were not executed. The annotator worked in an integrated development environment, studied the systems' structure beforehand, and maintained regular communication with the researchers to address any arising concerns. The collected traceability links were inspected and validated by one researcher, with another researcher verifying the links for at least ten test cases from each system.

Precision - the proportion of correctly detected test-code pairs is also calculatedm the formula can be observed on Equation 1.3. Here the upper part of the fraction denotes how many tests the algorithm could retrieve, while the bottom is the number of test cases that match the naming convention. This evaluation strategy is well suited for the listed systems since they are fairly well covered by proper naming conventions.

$$precision = \frac{|relevantTest \cap retrievedTest|}{|retrievedTest|} \tag{1.3}$$

With such an evaluation, it is only possible to find one pair to each test case correctly. The observed methods produce a list of recommendations in order of similarity. Every class is featured on this list. Thus, using this evaluation method, the customary precision and recall measures always coincide, which necessarily means that the F-measure metric would also have the same value. This is in accordance with the evaluation techniques commonly used for recommendation systems in software engineering. Because of this equality, I shall refer to these quantified results in the future as precision only.

Applicability of Naming Conventions

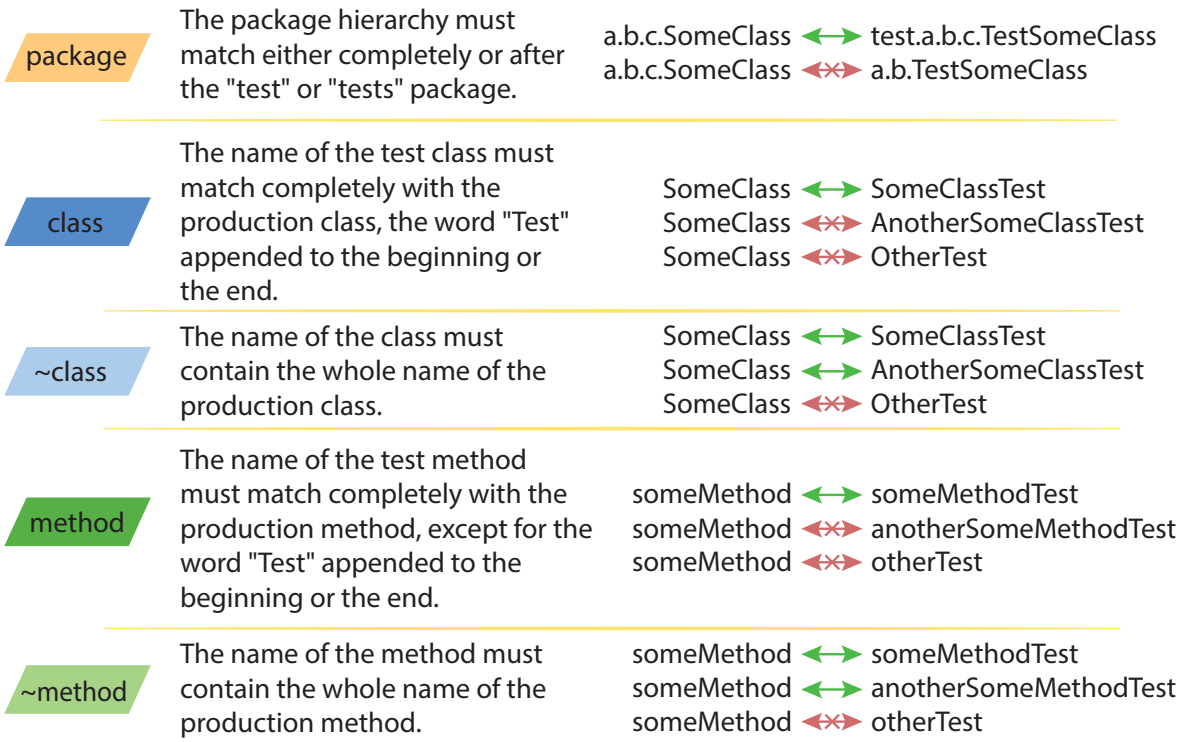


Figure 1.7: *Various possible naming convention criteria components.*

Naming conventions for tests encompass a wide range of practices and are often loosely defined. Typically, these conventions are informally agreed upon by developers, with written guidelines being rare. They are generally regarded as best practices, with usage varying across teams or even individuals. Given the diversity of naming conventions and their inconsistent application across different systems, flexible criteria are necessary to detect them effectively. We propose a set of general criteria for our examination, as illustrated in Figure 1.7. While other criteria, such as abbreviations or alternative identifiers for tests beyond the word ”Test,” are possible, the criteria presented here are among the most intuitive and commonly used. Let us consider some potential combinations of these criteria components:

- **P**ackage, **C**lass and **M**ethod = package + class + method
- **P**ackage and **M**ethod = package + method
- **P**ackage, **W**ildcard **C**lass and **W**ildcard **M**ethod = package + ~class + ~method
- **P**ackage and **W**ildcard **M**ethod = package + ~method
- **M**ethod = method

- **P**ackage and **C**lass = package + class
- **C**lass = class
- **W**ildcard **M**ethod = ∼method
- **P**ackage and **W**ildcard **C**lass = package + ∼class
- **W**ildcard **C**lass = ∼class

Some other viable combinations can also exist, which did not seem suitable for the unique distinction of test-code pairs. The criteria are ordered by strictness in a descending manner. While the stricter criteria produce more distinction between pairs, they are less versatile and are harder to uphold. Table 1.3 presents the extent to which the naming conventions were found to be applicable to the evaluated systems.

Table 1.3: *The applicability of NC using different approaches.*

Criteria	ArgoUML	Commons Lang	Commons Math	Gson	JFreeChart	Joda-Time	Mondrian	PMD
PCM	14.91%	17.04%	12.50%	1.74%	32.60%	3.60%	6.57%	7.93%
PM	20.73%	19.80%	16.85%	2.18%	38.95%	10.09%	9.04%	8.43%
PCWM	19.82%	56.67%	32.19%	9.59%	49.53%	23.78%	11.51%	15.86%
PWCWM	21.27%	66.73%	37.52%	9.59%	50.92%	59.65%	12.09%	16.48%
PWM	33.45%	70.79%	45.42%	15.47%	58.64%	74.42%	31.01%	25.53%
M	28.91%	19.96%	21.16%	3.05%	40.15%	11.38%	12.22%	11.90%
PC	60.18%	84.58%	75.07%	26.14%	96.47%	36.80%	17.82%	58.36%
C	64.00%	84.58%	75.07%	27.89%	96.47%	37.30%	20.81%	66.91%
WM	74.00%	80.00%	81.53%	60.24%	61.28%	78.55%	73.34%	58.36%
PWC	75.09%	99.11%	88.06%	28.87%	97.05%	98.04%	21.52%	61.09%
WC	80.55%	99.11%	91.42%	44.77%	97.41%	98.04%	35.96%	72.12%

1.6 Results

As detailed in the previous sections, five different source code representations and three text-based similarity techniques have been explored. We provide an overview of various naming conventions along with their relevance determined through automated extraction. Subsequently, we present our experiments employing the $ensemble_N$ approach, where we sought the optimal N value using NC-based and manual traceability links. Furthermore, we compare the traceability approaches based on both NC and manual assessments. It is important to note that production methods with fewer than three tokens in their bodies were excluded, as such trivial and abstract methods are unlikely to be the primary focus of testing.

1.6.1 NC-based Evaluation

Ensemble Experiments

In the Ensemble setup (detailed in Section 1.3.4) the relation between the used techniques is asymmetric, since the first methods ranked list refines the second ones. From experiment results it has been observed that combining LSI just with TF-IDF only seems to damage the results in both directions, the average precision using this combination was merely 50% for most similar classes, while 70% for the top five items in the ranked list. Combining LSI and TF-IDF with Doc2Vec in such a way that they provide the base of the ranked list and being only refined by Doc2Vec also performed poorly. However, the refinement of Doc2Vecs ranked list with both other techniques improved the results. The refinement of the similarity list with LSI resulted in more than 2.5% improvement compared to Doc2Vec as a standalone technique. Although TF-IDF also improved the values, the improvement was just under 2%. Due to space limitations and to stick to the results that seem more important these results are not displayed in detail. The combination of all three techniques was also explored: the main similarity list was provided by Doc2Vec and refined by TF-IDF and LSI. When Doc2Vec is combined with both of the other methods, the obtained results were even better than in the other cases. If only the most similar code class is considered, Doc2Vecs average precision improves by an absolute 3%.

Thus, the main similarity list is provided by Doc2Vec and it is refined with the other two algorithms, results are presented on Figure 1.9 and Figure 1.8. As one can see in the figures, the experiments were carried out using different N values: 50, 100, 200, and 400. These values only influence the size of each similarity list. If N is relatively big, then the filtering on the original similarity list (which originates from Doc2Vec) will not drop out many entries since many of the elements are present in the other two lists. In contrast, if N is a small number, the filtering is stricter since every similarity list contains only a limited number of entries. The previous argument can be further elaborated: if N is *big*, the resulting similarity list is going to rely mostly on the original one, while if it is *small*, the approach makes better use of the information from the other two lists.

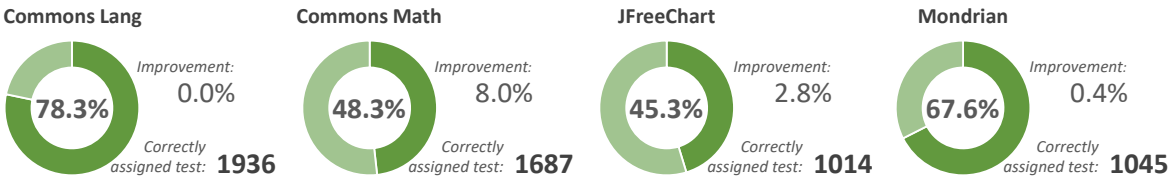


Figure 1.8: *Results showcasing Ensemble, trained on the IDENT representation of the source code.*

As depicted in the figures, the experiments were conducted with different *N* values: 50, 100, 200, and 400. These values solely influence the size of each similarity list. When *N* is relatively large, filtering the original similarity list (derived from Doc2Vec) will exclude fewer entries, as many elements are also present in the other two lists. Conversely, when *N* is small, the filtering becomes more stringent because each similarity list contains only a limited number of entries. To elaborate further, if *N* is large, the resulting similarity list will primarily depend on the original list. However, if *N* is small, the approach will make better use of the information from the other two lists.

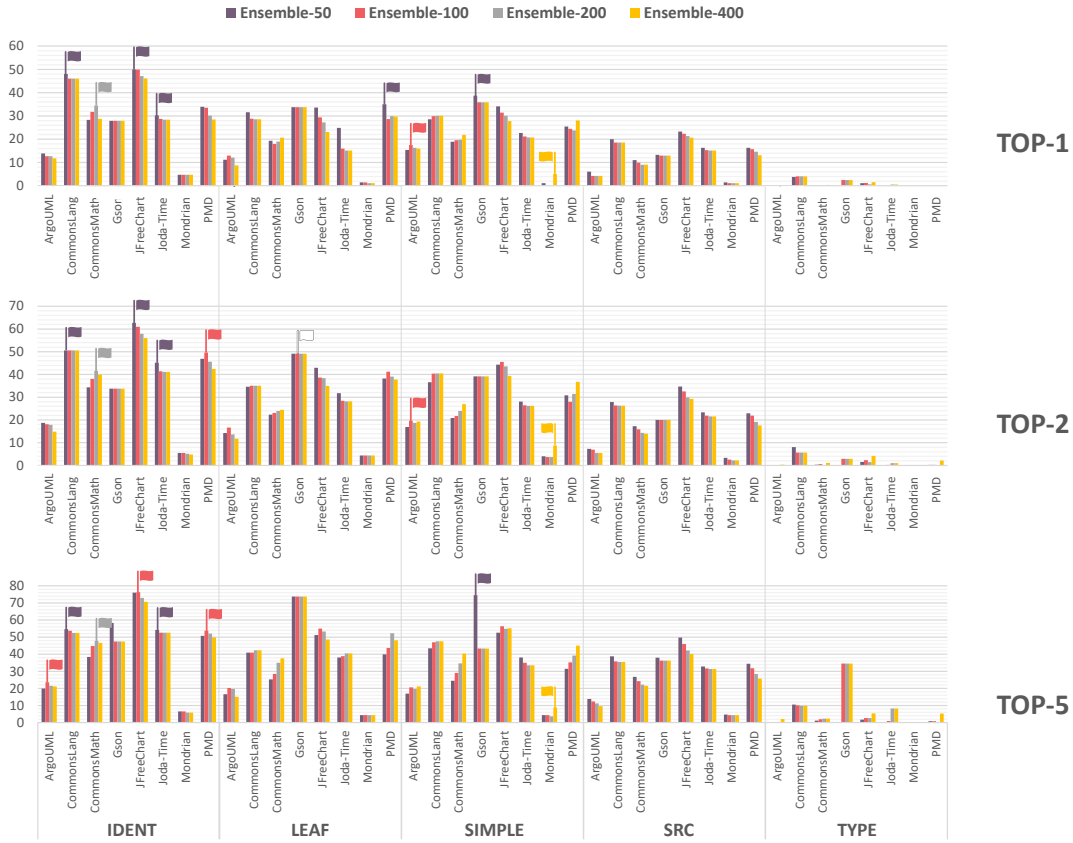


Figure 1.9: *Results of the $ensemble_N$ learning approach using NC-based evaluation.*

In Figure 1.9 the small flags atop the bars indicate the highest values for each system in their respective categories (top_1, top_2, top_5). The flag's color matches its corresponding bar, with a white flag indicating that the highest values are tied. Notably, there were no instances of two or three highest values being identical. This experiment also considers different source code representations. The figure shows that most flags appear at the IDENT representation. Additionally, the $ensemble_{50}$ approach tends to produce the highest values for top_1 results. For multiple recom-

mendations (top_2 and top_5), the results are less clear, with $ensemble_{100}$ also yielding good results. However, $ensemble_{400}$ appears to be less precise, being prevalent only in the case of Mondrian using the SIMPLE representation. The findings on the manual dataset further support this observation.

Based on the above observations, in the following sections, N is fixed to 50 in the $Ensemble_N$ experiments with Doc2Vec providing the main similarity list which is refined by the other two approaches.

Comparison of ML models

Table 1.4 shows the top_1 results of different machine learning approaches, evaluated via naming conventions. Cells of color teal indicate the highest values for each system within a method, while cells of color violet indicate the overall top values. For the $Ensemble_N$, only those cases are listed where N = 50, since this setting seemed to be the most beneficial (for further discussion see Subsection 1.3.4). The listed approaches correspond to the ones introduced in Section 1.3. The notion *[approach]+**CG*** refers to filtering with our soft computed call information described in Section 1.3.5. Based on the table, multiple observations can be made. First, the IDENT representation seems to fit the best for all of the listed techniques. It seems to capture just enough information, while still maintaining some level of abstraction. From the observed models, Doc2Vec predominantly resulted the best similarity lists. Moreover if the resulted similarity list is even enriched with CG information, the accuracy value goes up even further.

1.6.2 Evaluation on Manual Data

The results measured on the manual dataset are presented in Table 1.5. Similar to the previous table, teal indicates the highest values within a method, while violet highlights the overall highest value. Top-5 results are always equal to or higher than top-1 numbers, as multiple similar matches are considered during evaluation. This comparison involves different approaches against a dataset containing manually curated traceability links for four of our subject systems. Additionally, two new rows are introduced. The first row shows the applicability of the naming convention (denoted as PC in the previous subsection). These values reflect the applicability of the conventions to specific test cases in the dataset, rather than to the entire system.

If naming conventions are considered accurate, their precision should intuitively correspond to the achievable precision without relying on any additional IR-based approach, solely depending on the names. The last row, labeled with *NC*, represents our method, which first attempts to establish the link using naming conventions. If this fails, the Doc2Vec suggestion is considered. If the resulting precision values were

Table 1.4: *Top-1 results featuring the different text-based models trained on various source code representations, evaluated using naming conventions.* ■ *- highest value in a row* ■ *- highest value in a column*

Method	Representation	ArgoUML	C. Lang	C. Math	Gson	JFreeChart	Joda-Time	Mondrian	PMD
Doc2Vec	IDENT	19.63%	82.16%	50.00%	45.83%	49.22%	41.43%	66.42%	37.15%
	LEAF	18.43%	61.00%	33.01%	47.92%	25.10%	20.79%	65.33%	42.04%
	SIMPLE	24.77%	67.91%	33.78%	47.50%	30.69%	26.26%	65.33%	34.82%
	SRC	7.85%	31.32%	15.46%	16.67%	22.64%	22.30%	21.53%	15.92%
	TYPE	0.60%	4.36%	0.78%	2.92%	2.36%	5.18%	0.00%	0.00%
LSI	IDENT	32.93%	66.08%	19.42%	30.83%	33.29%	35.04%	22.99%	19.96%
	LEAF	14.80%	23.11%	3.63%	7.08%	9.63%	16.12%	11.31%	7.80%
	SIMPLE	15.71%	21.48%	3.47%	4.37%	13.15%	6.69%	4.38%	11.46%
	SRC	19.64%	54.64%	24.36%	14.17%	21.48%	28.20%	31.02%	22.29%
	TYPE	0.00%	0.48%	0.65%	4.58%	0.00%	0.50%	0.00%	0.00%
TF-IDF	IDENT	35.95%	73.62%	35.78%	35.00%	45.65%	48.71%	73.72%	24.63%
	LEAF	32.63%	70.94%	37.33%	38.33%	48.93%	47.77%	66.79%	23.99%
	SIMPLE	28.70%	69.49%	33.08%	30.00%	44.30%	47.77%	72.26%	23.57%
	SRC	27.79%	51.51%	28.68%	18.75%	25.19%	31.08%	50.73%	22.29%
	TYPE	0.00%	0.48%	0.65%	4.58%	0.00%	0.50%	0.00%	0.00%
Ensemble-50	IDENT	13.89%	48.00%	28.27%	27.92%	50.00%	30.22%	4.75%	33.97%
	LEAF	11.18%	31.61%	19.29%	33.75%	33.56%	24.89%	1.45%	35.03%
	SIMPLE	15.41%	28.54%	18.93%	38.75%	34.01%	22.73%	1.01%	25.48%
	SRC	6.04%	20.00%	11.02%	13.33%	23.24%	16.33%	1.46%	16.35%
	TYPE	0.00%	3.79%	0.12%	0.00%	1.11%	0.00%	0.00%	0.00%
Doc2Vec+CG	IDENT	45.01%	83.14%	61.04%	85.83%	62.82%	43.02%	68.61%	54.14%
	LEAF	42.29%	72.66%	45.65%	44.17%	56.48%	34.10%	73.72%	52.23%
	SIMPLE	41.69%	71.89%	51.41%	52.92%	58.06%	24.32%	74.09%	51.80%
	SRC	32.33%	54.48%	29.62%	35.42%	37.36%	23.38%	47.08%	36.31%
	TYPE	3.63%	17.61%	13.22%	42.08%	10.46%	16.04%	41.24%	15.92%

lower than before, it would imply that either the dataset is incorrect or the naming conventions are misleading. It is also evident that if the results of this combined approach and the pure *NC* approach were identical, the IR-based addition would be unnecessary. However, Table 1.5 shows that none of these concerns were realized. In fact, this combined approach produced the best results in almost every case.

1.7 Discussion

According to the results, the names of test *methods* are much less likely to mirror the names of their production pairs correctly. Although the experiments only deal with eight open-source systems, it is highly probable that the developers of other systems also tend to behave similarly in focusing more on class-level naming conventions. Production method names should be descriptive and lead to an easy to understand and quick comprehension of what the method does. This is also true about the names of test cases, they should also refer to what functionality they are aiming to assess. Consequently, the names of the test cases would become rather long if they

Table 1.5: *Top-1 and top-5 results featuring the different text-based models and the applicability of NC on each project. Models were trained on 5 different source code representations.* *- highest value in a row* *- highest value in a column*

Method	Representation	Top-1				Top-5			
		C. Lang	Gson	JFreeChart	Joda-Time	C. Lang	Gson	JFreeChart	Joda-Time
NC	-	76.00%	26.00%	0.00%	0.00%	76.00%	26.00%	0.00%	0.00%
Doc2Vec	IDENT	58.00%	15.69%	15.49%	32.00%	62.00%	25.49%	15.49%	48.00%
	LEAF	30.00%	13.73%	11.27%	20.00%	52.00%	21.57%	15.49%	52.00%
	SIMPLE	15.69%	17.65%	14.08%	16.00%	48.00%	21.57%	16.90%	52.00%
	SRC	16.00%	9.80%	12.68%	32.00%	42.00%	29.41%	30.99%	54.00%
	TYPE	4.00%	1.96%	11.27%	4.00%	22.00%	3.92%	11.27%	10.00%
LSI	IDENT	34.00%	17.65%	4.23%	10.00%	68.00%	5.64%	5.63%	44.00%
	LEAF	12.00%	7.84%	4.23%	2.00%	34.00%	23.53%	5.63%	28.00%
	SIMPLE	4.00%	5.88%	4.23%	2.00%	30.00%	23.53%	5.63%	24.00%
	SRC	34.00%	17.65%	12.68%	20.00%	70.00%	37.25%	23.94%	58.00%
	TYPE	4.00%	0.00%	0.00%	0.00%	8.00%	64.71%	0.00%	14.00%
TF-IDF	IDENT	30.00%	19.61%	4.23%	46.00%	76.00%	31.37%	5.63%	70.00%
	LEAF	30.00%	19.61%	4.23%	44.00%	76.00%	33.33%	5.63%	70.00%
	SIMPLE	28.00%	21.57%	4.23%	44.00%	72.00%	33.33%	5.63%	72.00%
	SRC	38.00%	19.61%	23.94%	12.00%	78.00%	43.14%	25.35%	68.00%
	TYPE	4.00%	0.00%	0.00%	0.00%	8.00%	64.71%	0.00%	14.00%
Ensemble-50	IDENT	44.00%	13.73%	4.23%	6.00%	52.00%	23.53%	4.23%	10.00%
	LEAF	13.73%	13.73%	4.23%	10.00%	38.00%	19.61%	4.23%	14.00%
	SIMPLE	14.00%	15.69%	4.23%	2.00%	40.00%	19.61%	4.23%	8.00%
	SRC	7.84%	11.76%	11.27%	12.00%	36.00%	17.65%	28.17%	22.00%
	TYPE	2.00%	1.96%	0.00%	2.00%	8.00%	1.96%	0.00%	2.00%
Doc2Vec+CG	IDENT	58.00%	64.71%	16.90%	24.00%	76.00%	80.39%	23.94%	64.00%
	LEAF	54.00%	54.90%	18.31%	20.00%	72.00%	78.43%	33.80%	66.00%
	SIMPLE	50.00%	56.86%	25.35%	26.00%	76.00%	78.43%	45.07%	64.00%
	SRC	50.00%	56.86%	36.62%	32.00%	78.00%	82.35%	66.19%	74.00%
	TYPE	42.00%	47.05%	11.27%	6.00%	62.00%	74.51%	28.17%	24.00%
Doc2Vec+CG+NC	IDENT	76.00%	64.71%	16.90%	24.00%	86.00%	72.55%	23.94%	64.00%
	LEAF	78.00%	64.71%	18.31%	20.00%	84.00%	70.59%	33.80%	66.00%
	SIMPLE	78.00%	66.71%	25.35%	26.00%	84.00%	76.47%	45.07%	64.00%
	SRC	80.00%	66.67%	36.62%	32.00%	88.00%	78.43%	66.19%	74.00%
	TYPE	74.00%	64.71%	11.27%	6.00%	78.00%	76.47%	28.17%	24.00%

always aimed to contain both the name of the method or methods under test and also provide additional meaningful information about the test itself. It can also be tough to properly reference the method under test on method level by naming conventions only. Polymorphism enables the creation of several methods with identical names that perform similar functionalities with different parameters. These should be tested individually, and test names can have a hard time distinguishing these. The inclusion of parameter types can be a possible solution as performed in Commons Lang for example, at the test case *test_toBooleanObject_String_String_String_String*, testing the production method *toBooleanObject* that gets four String parameters. Our manual investigation shows that test methods are indeed more likely to be named after the functionality they mean to test rather than after single methods even if they only test one method. One method can also be tested by multiple test cases. Thus this is not a very surprising circumstance. It is apparent that naming conventions on the method

level have to be more complicated, and their maintenance necessitates more work on the part of the developers. Thus, method-level naming solutions are likely to be a less valuable option in method-level test-to-code traceability. On the other hand, method-level traceability still requires proper class-level traceability. Thus, names should still be helpful.

Although serious differences can be observed between systems, method-level naming conventions are either complicated or entirely abandoned in most cases, which means that their usefulness is negligible in a general extraction algorithm. Class-level naming conventions seem to be better regarded by developers, and there is a visible effort to uphold them. Our findings show them to be suitable for general use in automatic extraction algorithms. Matching package hierarchies do not provide precise results but seem to be at least as commonly used as class-level conventions. They are likely to be suitable for filtering out false-positive results in algorithms.

1.7.1 Traceability Link Recovery Technique Improvements

It is immediately evident that the teal cells are predominantly located in the first rows of Table 1.4. The IDENT source code representation is notably prevalent, achieving the highest values in 37 out of 48 cases (77%). The violet cells are confined to the last vertical segment of the table, specifically in the Doc2Vec+CG approach. The $Ensemble_{50}$ approach yielded better results than standalone techniques (Doc2Vec, LSI, TF-IDF), and incorporating soft-computed call graph information further improved these results. Doc2Vec supplemented with call graph information achieved the highest precision values. This is likely because $Ensemble_N$ acts as a filtering technique, producing a reduced similarity list compared to the original, especially when *N* is small. Consequently, $Ensemble_{50}$ might exclude some correct links even before applying the call graph (CG) information.

According to the results in the table, IDENT is the most precise approach, except for the Mondrian project, where the SIMPLE representation performs best. The difference between IDENT and SIMPLE, however, is not significant. Notably, in cases where IDENT is not predominant, SIMPLE is best in 5 out of 11 instances. IDENT and SIMPLE are quite similar, which is reflected in the results. In contrast, TYPE consistently produces weaker results across all approaches. LEAF is also less precise, likely because its structure significantly overlaps with TYPE (LEAF combines IDENT and TYPE). This suggests that the TYPE information in an AST holds less significance for the text-based test-to-code traceability task.

Our inspections concluded that Doc2Vec seems to be the best-performing standalone technique in the field. Although combinations of different techniques can also boost the results, the textually extracted soft-computed call information is likely to boost IR-based approaches even more. In a scenario of combined techniques, call graphs can be a valid filter even for textual connections.

1.7.2 Performance on Manual Data

Compared to the NC-based evaluation, the results from the manual dataset are more complex to interpret. As shown in Table 1.5, not every violet cell appears in the last row, but most do. Analyzing the top-1 results on the left side of the table, we find that in 3 out of 4 systems, the highest precision values are achieved using Doc2Vec combined with call information and naming conventions. The exception is Joda-Time, where TF-IDF is predominant. However, TF-IDF results tend to vary more than others, suggesting this individual case may be due to chance. Nevertheless, Doc2Vec+CG still produced high precision values, and the inclusion of naming conventions further improved the approach. For JFreeChart and Joda-Time, the results did not improve with the added naming convention pairs, which is expected since the dataset intentionally contained links not covered by naming conventions. This indicates that IR-based approaches can successfully supplement naming conventions while maintaining their beneficial properties.

On Table 1.5 the precision values are higher, which is expected as the text-based models have a broader range to identify correct matches. While the top-1 results varied in precision, with JFreeChart and Joda-Time having lower results, even a small number of additional candidate links significantly improved match correctness. Further experiments showed that considering top-10 or top-20 results often yielded a 100% match, although developers are unlikely to search through lists of this length in practice. Therefore, a recommendation system providing five results could still be practical.

From the manual dataset analysis, it is evident that in projects with proper naming conventions, traceability links can be extracted based solely on this information. However, for projects lacking such conventions, IR-based techniques can identify the correct links in a significant number of cases. Comparing the results of the final Doc2Vec approach to NC alone, the precision values increased by an average of 28%. Even with complex, system-specific naming conventions, IR-based methods can provide substantial assistance. Good programming practices can also enhance the performance of text-based techniques, which rely on names more flexibly.

The choice of an ideal input representation is less clear-cut than in the previous case. While the IDENT representation was predominant in the NC-based comparison, the SRC representation yielded the highest precision values here. Among

AST-based representations, SIMPLE performed best. The SRC representation also excelled in the Top-5 values, which is advantageous as it is purely text-based. Since the Doc2Vec+CG+NC method uses call information extracted via regular expressions, it remains viable without static analyzer tools. While both IDENT and SRC contribute valuable data, the variation in their performance across different datasets suggests that further research is needed to determine the best representation. Possible errors in the automatically gathered NC data and the limited size of the manual dataset could affect these findings, necessitating additional investigation.

According to the manual data, Doc2Vec achieves the best results in most cases. In exceptional cases, however, other text-based techniques can still outperform it. The use of naming conventions and call information also tends to improve the results further. Naming conventions, if existing, are highly precise and can be supplemented with other IR-based techniques to achieve a more versatile text-based approach.

1.7.3 Implications

Our experiments yield several key insights for researchers and developers aiming to build new test-to-code traceability systems. While naming conventions are highly precise when applied correctly, their implementation at the method level is challenging, as the source code often lacks sufficient connections that can be extracted through simple rules. However, package and class names can effectively imply connections, even at this level, where method names may not be as informative. Thus, naming conventions are extremely useful across all levels of traceability link extraction, and incorporating them into new extraction methods would likely enhance their performance.

Doc2Vec represents a significant advancement over more traditional semantic similarity techniques. Although it is slightly more resource-intensive than LSI, the difference is not substantial, and like other techniques, Doc2Vec can provide real-time results for identifying the most similar parts of the code for a given test case. Therefore, if only one textual technique is to be considered, Doc2Vec is the optimal choice.

Combining Doc2Vec with other techniques can yield even better results. However, using other textual techniques as filters can sometimes exclude valuable data, potentially negatively impacting the overall performance when combined with non-semantic techniques. Call information, even when gathered via regular expressions, significantly enhances these techniques. Combining call graphs obtained through static or dynamic analysis could further improve precision, as evidenced by current state-of-the-art solutions where the LCBA (Last Call Before Assert) technique is considered one of the most reliable methods.

The effectiveness of different source code representations remains inconclusive. While IDENT performed best in the NC-based evaluations, manual data suggests that SRC contributes most to correct extraction. Therefore, further experiments are required to determine the optimal representation definitively. Nevertheless, IDENT and SRC are the most promising options.

1.8 Threats to Validity

Although our experiments were conducted with the intention of providing a large-scale evaluation and a relatively deep comprehension of current textual methods, some threats to the validity of the derived conclusions still have to be mentioned. While naming conventions are considered a very precise source of information, they have clear limitations. Thus, our automatically-collected evaluation data may contain some errors and is likely to miss at least some valid links. Although manual data is usually considered best, naming conventions enabled us to assess hundreds of tests for each system and even thousands for most. On the other hand, our manual dataset used for the evaluation is limited in size. Thus, noise in the data could cause discrepancies in the results. This could be tackled by the inclusion of additional manual data, which will hopefully be more widely available in the future.

Our experiments only covered systems written in the Java language. This is a significant limitation as Java differs greatly from several other popular programming languages. Even the structure of the code can show severe differences. Popular naming conventions can vary in these circumstances, new viable combinations could be constructed, and others could become less relevant. This also reflects a great amount on the source code representations. Even the text and even variable names could be susceptible to such a difference. On the other hand, textual methods, building on semantic information rather than program structure, are still the most likely to retain their properties this way.

The experiments were conducted on JUnit tests. The JUnit framework is one of the trail-blazers of current software testing and is extremely popular among developers. Still, it is easy to see that other tests could perform differently when subjected to the experiments. Even in this, however, semantic information should be the least affected as it does not rely on a specific structure or specific forms of assertion statements.

Similarly to the difference in programming languages, the size of the systems could also influence the results. Our systems under evaluation are all medium-sized open-source systems. There is no guarantee that small or large systems would perform the same way, even though the question of proper traceability is probably easier for small systems. The same questions can arise about the domains of the systems, which could also affect traceability. It is visible that systems vary significantly in

their properties. An average value of precision is thus hard to pinpoint, it is easier to compare techniques to each other. Our experiments covered more than 1.25 million code lines to provide a large-scale investigation, significantly more than our previous inquiries.

Our experiments with naming conventions and even the source code representations represent the options we found most viable. There might be many more naming conventions that could be applied to some systems with great success, even with automated extraction. As there are usually no descriptions about naming conventions for software systems, finding these and judging their usefulness is highly complex. Our experiments considered some of the most simple and widely used conventions. There seems to be a balance between complete precision and easy usage in naming conventions. Our experiments also attempted to investigate this, building our subsequent experiments on a middle way that seemed widely applicable but still precise for our current level.

1.9 Concluding remarks

Test-to-code traceability helps to find production code for a given test case. Our assumption was that the related test and code classes are similar to each other in some sense. We employed three different similarity concepts, based on Doc2Vec, LSI and TF-IDF. Since these methods are intended for natural language texts, we experimented with three different source code representations. Analyzing the obtained data, we derived the conclusion that from simple source code representations, IDENT performs more desirable in test traceability. We compared the obtained results from the three textual similarity techniques and found that the Doc2Vec based similarity performs better in the recovery task than other approaches. Finally, we refined Doc2Vec's ranked similarity list with the recommendation of the other approaches. With this experiment we have successfully improved the performance of Doc2Vec for every project, therefore introducing a successful mixed approach for the textual matching of tests and their production code.

In the preceding sections, experiments focused on the textual aspect of enhancing test-to-code traceability. Two prominent techniques, reliance on naming conventions and information retrieval, were explored, with new ideas, experiments, and observations provided on their potential improvements and combination opportunities.

We conducted a detailed investigation of developers' naming convention practices through experiments with eight open-source systems and nine possible combinations of generalizable and simple rules. This experiment revealed that package- and class-level conventions are generally followed with at least moderate effort, whereas method-level conventions, although present in every system, are less consistently upheld. In addition to our evaluation on manual data, we used an automatic extraction

method for further evaluation, relying on package- and class-level conventions.

The evaluation encompassed six traceability link extraction methods assessed with five distinct source code representations. Among these, the identifier-centric (IDENT) representation, leveraging ASTs, emerged as the most effective in the majority of cases during the naming convention-based evaluation. However, the text-centric (SRC) representation exhibited greater precision when compared to a limited amount of manual data. The inclusion of call information retrieved via regular expressions significantly bolstered results when employed as a filtering technique for Doc2Vec. Although Latent Semantic Indexing (LSI) and Term Frequency-Inverse Document Frequency (TF-IDF) also demonstrated promise for the same purpose, the fusion of Doc2Vec and call information yielded superior results. While well-defined and consistently adhered to naming conventions have the potential to yield highly precise traceability links, their application remains constrained. Automatic recovery of naming conventions stands to benefit substantially from the integration of other text-based techniques, fostering a versatile semantic approach that can be effectively employed alongside other mainstream methods.

The author of this PhD thesis is responsible for the following contributions presented in this chapter:

I/1. Implementation of the following models: Doc2Vec, TF-IDF and $Ensemble_N$.

I/2. Design and implementation of the source code representations.

I/3. Design and implementation of the Call Graph information extraction technique from source files.

I/4. Planning, execution and explanation of the experiments.

I/5. Measurement and visualization of the evaluation metrics.

2 Machine Learning in Automated Program Repair

2.1 Overview

In recent years, there has been growing interest in using Machine Learning techniques for Automated Program Repair [109]. These techniques have shown promise in generating high-quality repair patches for a variety of programming languages and domains [128]. However, APR is a challenging task due to the complexity and variability of software systems with many open challenges [52, 96]. Generating repair patches that are both correct and maintain the original functionality of the program is a non-trivial task. APR approaches may be limited by the quality and coverage of the training data, as well as the ability of the model to generalize to new programs and defects. Additionally, APR approaches must be able to scale to large codebases and handle a wide range of defects. As a result, there is a need for further research to improve the effectiveness and efficiency of these approaches [42, 141, 204].

Many flagship APR solutions are implemented in such a way to repair programs written in C, Java or even Python. Since the appearance of GenProg [174] and it's genetic approach, many excellent researchers tried to improve the performance of it by creating several distinct APR tools [63, 114, 116, 119, 145, 169, 190]. Many of these follow the Generate-and-Validate (G&V) approach [58], that is, first a patch is generated and than the test suite is executed to check the correctness of the generated candidate. However, the previously mentioned programming languages are frequently used for desktop and android applications, or in research projects, their use is not widespread in web development. For the seventh year in a row, JavaScript (JS) is the most commonly used programming language [162]. It is the de-facto web programming language globally and the most adopted language on GitHub [56]. JavaScript is massively used in the client-side of web applications to achieve high responsiveness and user friendliness. In recent years, due to its flexibility and effectiveness, it has been increasingly adopted also for server-side development, leading to full-stack web applications [77].

Data-driven Automated Program Repair (APR) approaches utilize Machine Learning techniques to learn from a dataset of programs and their corresponding repair patches. One advantage of data-driven APR approaches is that they can learn from a variety of repair patches, allowing them to adapt to different repair strategies and programming languages. These approaches can be effective in generating high-quality repair patches for a wide range of defects and programming languages [113, 196]. Data-driven APR approaches often involve training a Machine Learning model on a large dataset of programs and their corresponding repair patches, and then using the trained model to generate repair patches for new programs with defects. One such dataset was made in 2019 by Tufano *et al.* [168], on which these models can be trained and evaluated. Their seminal work has been encased in the CodeXGLUE benchmark [111], featuring a platform for future publications including diverse programming language tasks. The dataset is highly successful among researchers, new model architectures are proposed rapidly (usually published on arXiv.org first, thus bypassing the traditionally slow publication process) and new records are booked in a monthly basis.

The current chapter applies ML on several subtasks in the APR domain, Figure 2.10 depicts a comprehensive overview of it. Since fixing a bug always starts by localizing it, in Section 2.3 we dive into more details about Fault Localization. The main focus in this chapter is on the relationship between traditional and Deep Learning algorithms. By conducting a large-scale training, the stability of DL-based FL methods are investigated. As can be seen on Figure 2.10, the FL part of the process takes the coverage information from test execution and outputs a list of most suspicious statements that needs to be repaired. The classical G&V patch generation approaches then try to modify these faulty statements to fix the whole program. On the other hand, DL approaches usually refrain from FL (assuming perfect knowledge) and only focus on the generation of repair candidates based on a dataset. Data flows for each approach are marked on Figure 2.10: orange arrows signal data for traditional APR approaches, blue arrows for learning-based approaches, while flows that are employed by both of them are marked with black arrows.

Next, in Section 2.5 a data collection approach is described, since modern data-intensive APR applications need a lot of training data. As JavaScript lacks such curated dataset, the evaluation of the proposed APR methods is difficult. The section introduces the *FixJS* dataset, describes its properties and structure. The proposed dataset is available on GitHub [2] and have a DOI to make it easily citable [3]. It contains roughly two million bug-fixing commits from GitHub. From these commits, the modified functions were extracted (the state before and after the bug fix happened). These functions are then tokenized and abstracted, resulting in three different source

[2] `https://github.com/RGAI-USZ/FixJS`
[3] 10.5281/zenodo.6340207

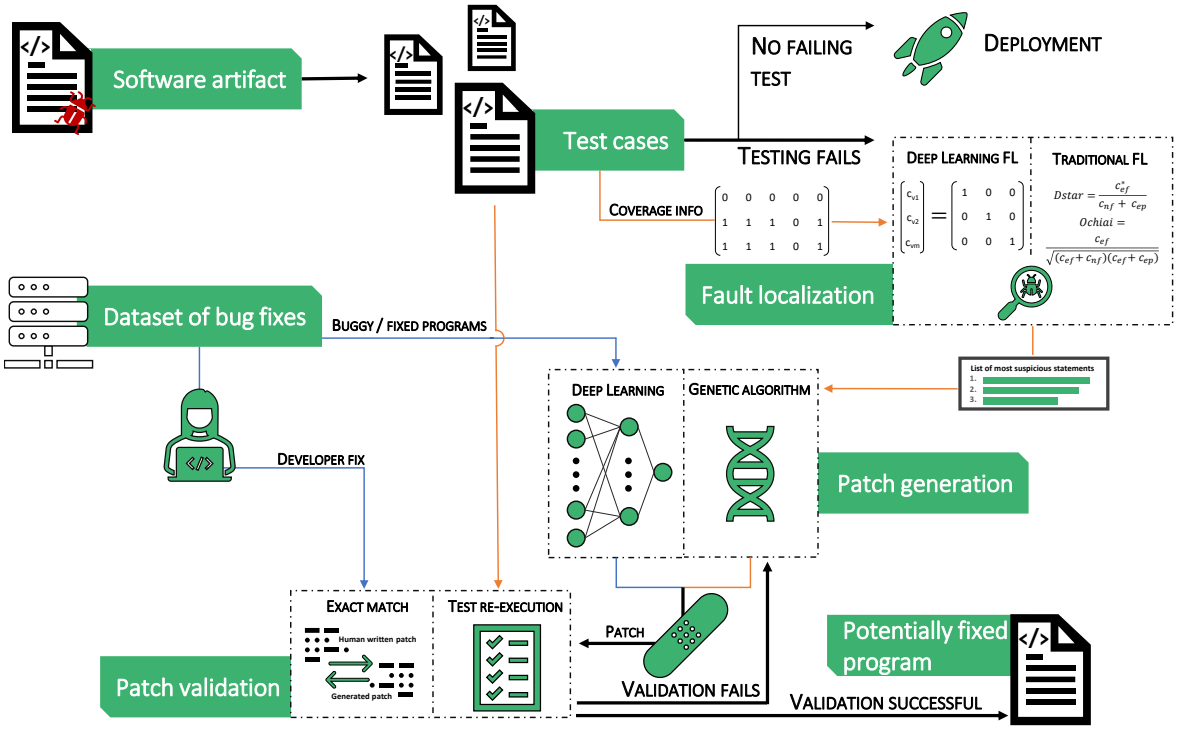


Figure 2.10: *A comprehensive overview of Theses II.*

code representations. FixJS can be used to train and evaluate a deep learning model that predicts correct fixes without any further processing steps.

The remaining of the chapter focuses on the generation of repair patches both with traditional and data-driven approaches. An adaptation of the seminal APR tool GenProg has been adapted to JavaScript. Also, to bypass the cumbersome process of designing, training, and evaluating a new model, a new approach is proposed that fixes buggy programs automatically using ChatGPT, simply relying on this Large Language Model (LLM). On Figure 2.10 we can see that there is a fundamental difference of the two: while learning-based APR approaches are evaluated against the developer fix (i.e. the automatically generated patch should match the one created by human developer), classic G&V tools evaluate the generated patch by re-executing the test cases. The former one is obviously more strict, but has three main flaws: (1) it needs a sufficient database to train and evaluate on (see Section 2.5), (2) does not consider semantically identical patches and (3) overlook the fact that a developer patch can be flawed. On the other hand, classifying a patch by test execution is both more time-consuming (due to the execution of the tests) and more susceptible to overfitting - where only test cases are successfully executed, but the real bug is not fixed. This phenomena is detailed in Chapter 2.9. As the repair process usually starts with Fault Localization, after depicting an overview of related works, in the next chapter the conducted work on this domain is described.

2.2 Related Work

Classical APR approaches

There have been several implementations for Java and C programming languages which aimed at automatically repairing programs. GenProg [174] was one of the first to perform a fully automatic fix with relatively good results. It was originally written for the C programming language, but has since been implemented for Java [116]. The tool uses genetic programming (GP) to fix errors without the need for any formal specifications. PAR is a synthesis-based tool which leverages the knowledge of human-written patches [87]. It works in Java and repairs source code based on 10 predefined templates. DeepFix is another repair tool, which uses a multilayer neural network. The program to be repaired is considered as a sequence of tokens from which another is generated with appropriate transformation [63]. In addition to completely general repair techniques, there are those that specialize only in certain error classes. An example is Nopol, which improves conditional control structures (if-then-else structure). In the generated patch Nopol either modifies the if structure or gives it a prerequisite (guard condition) [190]. Another such tool is Kali, which uses only deleting or skipping the source code to synthesize patches. This is obviously not a program repair technique, it is mainly used to identify weak test cases and poorly specified bugs [145]. Figra has been proposed in a recent work [51], where the tool improve the fix step through a search-based automated program repair technique that reuses code from the program under repair. Their results yield that Figra outperforms other APR tools in terms of correctness on the observed benchmarks. In a 2014 initiative, a framework was created that can automatically repair Java programs [117]. The name of the framework is Astor and its goal is to combine previous approaches and provide a unified interface for the APR community. It also includes the implementation of several repair strategies, such as Genprog, Kali or Cardumen. Of course, there are other approaches that tend to generate the fix from previous manual fixes [87, 95]. Although such approaches have yielded promising results, they have been criticized several times [126].

However, a general repair tool for JavaScript does not exist, there are several works that are closely related to this topic. In a recent work [151] authors presented Mutode, a generic mutation testing tool. In the paper several mutation operators are defined, from which most of them are pretty straightforward and not specified for the JavaScript language. The search for genetic operators has a long history, and many papers proposed several general operators. For instance in [97] authors investigate representation and operator choices for evolutionary program repair in the original GenProg framework. Another interesting approach is presented by Jensen et al. [77] where the authors automatically transformed common uses of *eval* into other lan-

guage constructs. While their work is not considered directly as Automatic Program Repair, the paper is an excellent basis for further research. The paper of Selakovic et al. [158] also focuses on JavaScript bugs, but their approach is not general because the presented tool only fixes performance bugs using static patterns.

Data-driven Repair Approaches and LLMs

Data-driven repair approaches form a separate research area in the Automated Program Repair field [128]. These techniques usually create a large train-test-validate database and evaluate the tool on that. Recently several such tools have emerged, such as SequenceR [25], Hoppity [41], DLFix [196], CoCoNuT [113] or CURE [80]. Hoppity predicts the changes to be made to the AST of JavaScript commits with a graph-based neural network. Transformer-based approaches seem to dominate this subfield recently, a great amount of work use this model to synthesize source code. For code completion Transformers also [88] perform great by learning the syntax of the language by including AST information to the input/output sequences. Finding the right source code representation is often not trivial for learning-based approaches, a recent study tries to tackle with this challenging problem [130]. In a recent work [24] Chen *et al.* addressed the problem of automatic repair of software vulnerabilities by training a Transformer on a large bug fixing corpus. They concluded that transfer learning works well for repairing security vulnerabilities in C compared to learning on a small dataset. Variants of the Transformer model are also used for code-related tasks, like in [3] where authors propose a grammar-based rule-to-rule model which leverages two encoders modeling both the original token sequence and the grammar rules, enhanced with a new tree-based self-attention. Their proposed approach outperformed the state-of-the-art baselines in terms of generated code accuracy. Another seminal work is DeepDebug [42], where the authors used pretrained Transformers to fix bugs automatically.

Prenner *et al.* [144] used Codex for automated program repair, evaluating it on the QuixBugs benchmark [106], consisting of 40 bugs in Python and Java. Although they experimented with different prompts, their focus was primarily on code generation from docstrings. Although OpenAI has introduced GPT-4 recently [132], the available scientific literature of it is is scarce. There is no available scientific paper specifically for ChatGPT and most of its use cases are undocumented in scientific literature, her a brief review is presented on the use of the GPT family. GPT-2 [8] was introduced in 2018, followed by GPT-3 [19] in 2020 by OpenAI. They have been applied to various tasks, including poetry, news, and essay writing [44, 202]. The capabilities of GPT have also been explored in the CodeXGLUE benchmark [111] for multiple tasks, where CodeGPT achieved an overall score of 71.28 in the code completion task. In a recent work [7], CodeGPT served as a baseline model for text-to-code and code generation tasks. A recent work introduces Text2App [66], which enables the creation of functional applications from natural language specifications.

Deep Learning Fault Localization

In the realm of DLFL, coverage matrix-based approaches have emerged as pivotal contributions. Zhang *et al.* in their seminal work [201] utilized three deep learning architectures (MLP, CNN, RNN) for fault localization. Subsequent research expanded on this foundation, employing oversampling [200] and test generation [199] to address class imbalance. The results demonstrated that these approaches enhanced fault localization effectiveness, surpassing previous DLFL approaches. The group later proposed Aeneas, synthesizing failing test cases from a reduced feature space [184]. This approach statistically outperformed baselines. To address class imbalance further, subsequent works introduced cost-effective data augmentation approaches, such as between-class learning [100] and the Lamont approach [72]. Additionally, the use of Generative Adversarial Networks (GANs) in CGAN4FL [101] demonstrated their efficacy in constructing a class-balanced dataset for fault localization. While these publications share a common theme, each introduces a unique improvement in fault localization. Unfortunately, not all listed papers have online appendices or repositories, but only one [184] provides source code. Also, the lack of predefined seeds in the training process poses challenges to the reproducibility of their experiments.

The issue of data imbalance in intelligent fault diagnosis methods has garnered extensive attention, leading to numerous publications [104, 110, 149]. For instance, Fang et al. [47] employ a conditional variational autoencoder (CVAE) for synthesis and apply fault localization techniques. GNet4FL [146] combines static and dynamic features for more precise fault localization. Other techniques, such as DeepFL [103] and FLUCCS [161], leverage additional information, with DeepFL automatically learning latent features and FLUCCS using Genetic Programming and linear rank Support Vector Machines (SVMs) for learning fault localization formulae.

Stability of Deep Learning

Conventional training methods for neural networks incorporate various sources of randomness, such as initialization, mini-batch order, and data augmentation. Since neural networks tend to be significantly over-parameterized in practical applications, this inherent randomness can lead to issues [75]. Situations when two models independently trained by the same algorithm produce differing predictions for the same input are referred to by literature as *churn* [14, 27]. Churn represents the proportion of test samples where predictions of the models do not match. One approach to mitigate churn involves eradicating all forms of randomness within the training configuration. However, even if one manages to control the seed used for random initialization and the data ordering, which itself presents challenges, it remains difficult to evade the inherent non-determinism present in contemporary computing platforms [129]. Distillation [78] transfers knowledge from larger to smaller neural networks to reduce churn, while this paper does not explore other properties of neu-

ral networks [90]. Having stable models capable of generating predictions unaffected by the random training factors is essential for developers to trust DL approaches - a challange modern ML still have to tackle with [60].

Liu *et al.* [108] examines the reproducibility of DL models in the field of SE. The study reveals that only 10.2% of the reviewed publications address replicability or reproducibility, with over 62.6% not sharing high-quality source code or complete data. Experimental results underscore the importance of reproducibility and replicability, demonstrating challenges in reproducing DL model performance due to an unstable optimization process, non-convergence in model training, and sensitivity to vocabulary and testing data size. While sharing a reproduction package in deep learning supports reproducibility, the inherent randomness in model initialization and optimization makes it challenging to guarantee that models trained by different researchers will produce identical experimental results even when re-running the provided source code and data. Thus stable models are fundamental to have reliable and applicable DL models in any domain.

2.3 Coverage Matrix-Based Fault Localization

To the best of my knowledge, DL-enhanced coverage matrix-based fault localization was introduced in 2009 by Wong *et al.* [182]. In tthe following their proposed method is described in short. Suppose we have a program with *m* executable statements and exactly one fault. Suppose also that there are *n* executable test cases of which *k* tests are successful and *n - k* are failed. This data is then organized into an *n x m* sized matrix, where inside the matrix there is 1 if the test covers the statement and 0 otherwise. An example of such a matrix can be seen in Figure 2.11. The result of the test execution is also organized into a vector form that has *n* rows, denoting whether the test was successful (0) or failed (1).

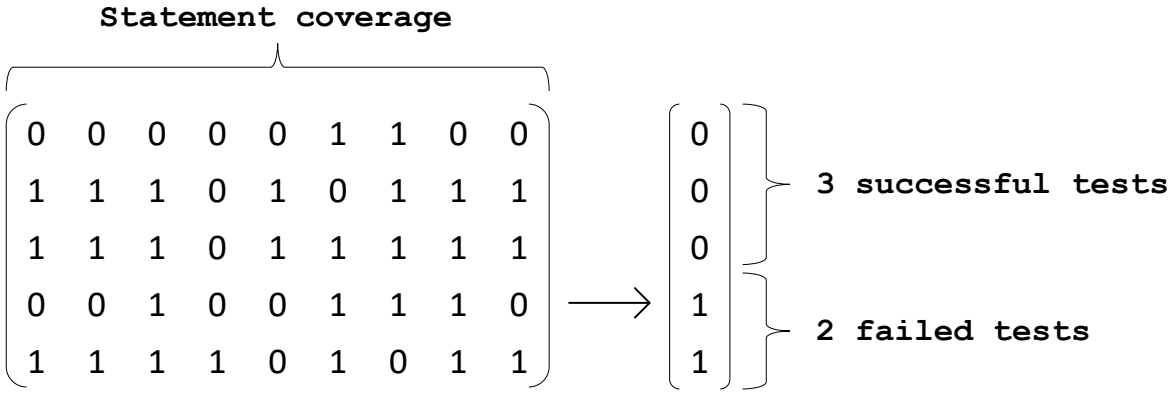
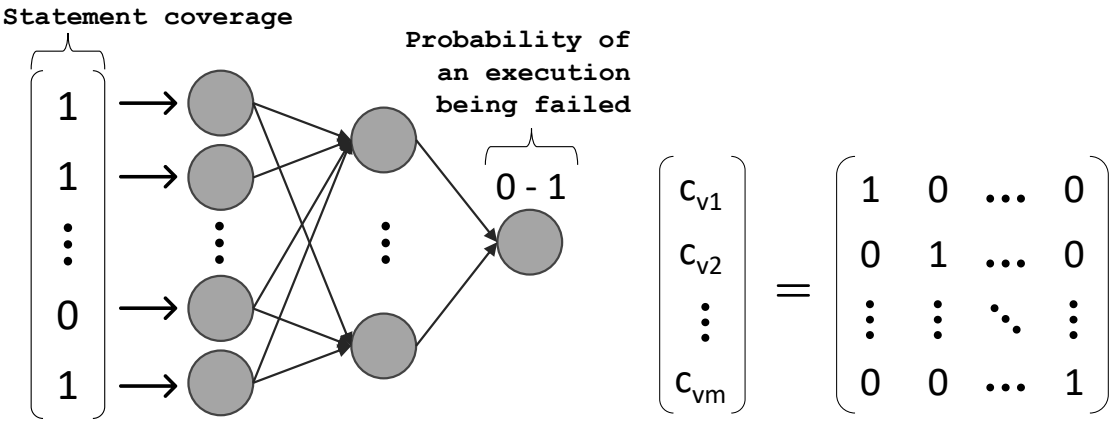


Figure 2.11: *Coverage matrix with 9 statements and 5 test cases.*

Next, a neural network is being constructed. The input layer has m neurons, as a single row from the above matrix (*i.e.* 'statement coverage vector') forms the input of the model. The output is a single neuron, and the desired outcome is to predict whether the test execution result was successful or not. The inner structure of the neural network is arbitrary, in the literature one can find diverse architectures, including Multi-Layer Perceptron, Convolutional Neural Network, Recurrent Neural Network and even Graph Neural Networks [124, 146, 181]. The outcome can be interpreted as an estimation of the test execution result. Note that there is no train-test-validation split of the training data, as the training objective is to learn to predict whether a test was successful or not, based only on the test coverage matrix as we depicted this process on Figure 2.12 (a). Thus the training objective is to perfectly learn when will a test case fail and when will succeed, no generalization needed.



(a) *High-level illustration of the neural network used for fault localization.* **(b)** *Virtual test simulating coverage of a single statement.*

Figure 2.12: *Components of DL-enhanced SBFL.*

The fault localization part comes after the neural network has been trained. Assume there is a set of so called *virtual test cases* whose coverage vectors are $c_{v1}, ..., c_{vm}$. The execution of a virtual test case covers only one statement and if we organize these into a matrix form again, the result is a diagonal matrix as shown in Figure 2.12 (b). The execution of such a virtual test case is interpreted as a *test that only covers one statement*. If a statement is contained in a lot of failed test executions, the output of such a virtual test case is expected to be high. This implies that during the fault localization, we should first examine the statements whose output values are high. The output value of the neural network is between 0 and 1, the larger the value is the more likely it is that the corresponding statement (in which in the coverage vector the value was 1) contains a bug. This output can be treated as the suspiciousness of a given statement in terms of its likelihood of containing the bug.

The fault localization process continues from this step the usual way: the statements are ranked based on their suspiciousness - more suspicious statements are sorted at the top, while less suspicious ones to the bottom of the list.

2.3.1 Measuring Model Stability using Churn

Here, churn is interpreted as defined by Cormier *et al.* as the expected disagreement between the predictions of two models [27]. Churn is zero if both models provide the same output for the same input, while large churn values mean that models independently trained disagree on most of the test data - suggesting that an error could have occurred in model design, or in learning and evaluation phase. This probability is essentially implemented in practice as:

$$1 - \frac{\#SamplesOnModelsAgreed}{\#AllSamples} \tag{2.4}$$

where $\#SamplesOnModelsAgreed$ is the number of samples on which the two independently trained models agree, while $\#AllSamples$ is the number of samples in the dataset. In Figure 2.13 one can observe that most of the assigned ranks differ between independent trainings.

2.3.2 Adapting Churn for Fault Localization

The straightforward adaptation of this metric for fault localization would be to consider every rank as a class label and models need to produce the exact same list of suspicious statements. However, two key observations can be made: (1) statements on similar positions are counted as disagreement and (2) not all ranks are of equal importance - buggy statements are more important. To overcome these limitation we define two revised versions of churn for fault localization.

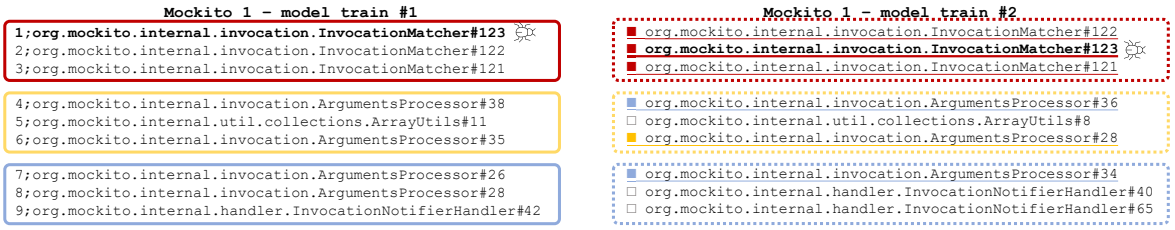


Figure 2.13: *Statement boxing including 3 statements each. Boxes are highlighted with colors, the faulty statement is bold and the list is ordered by the suspiciousness assigned by each model. The churn is calculated as follows:* $Churn = 1 - 1/12 = 0.917$*,* $BoxChurn = 1 - 5/12 = 0.583$*, while* $FlBoxChurn = 1 - 1/1 = 0$*.*

Statement Boxing: in the following, *boxing* is referred to as the process that groups statements into fixed sized boxes and then churn is measured not on the actual ranks, but on the assigned boxes. For example, in Figure 2.13 we defined a box size of 3. It can be observed that although the two independent trainings placed the faulty statement on different ranks, they are mapped in the same box, thus reducing churn value. It is expected that $BoxChurn$ values are lower compared to regular $Churn$, as fewer classes are created in this new setting.

Faulty Box: as the rank of the faulty statement is the most important in fault localization applications, we decided to adapt churn to meet this criteria. In the previous metrics disagreement between correct statements is also measured. To overcome this limitation, $FlBoxChurn$ is defined as $BoxChurn$ calculated only on the box containing faulty statements. Note that there might be more faulty statements in a system, thus the value of $FlBoxChurn$ is not necessarily 0 or 1. In Figure 2.13 this amendment results in a churn value of 0, which gives a better indication of the problem.

SFL Metric - Expense SBFL's effectiveness can be measured in various ways [67], but most rely on estimating the effort programmers need to identify the faulty element using the tool. The rank list serves as a proxy for this property, with the number of elements before the first faulty element, often collectively called the *Expense*. Most often, the absolute version of the Expense metric is used which means that we simply count the number of code elements in the rank list in front of the faulty one. One complication with this method are rank ties [189], i.e. situations when different code elements share the same suspiciousness scores. Typically, all elements in a rank tie are assigned the same rank value, based on one of these approaches [183]: *minimum*, which refers to the top most position of the elements sharing the same suspiciousness value (optimistic or the best case); *maximum*, where the bottom most position is used (pessimistic or the worst case); and the *average* strategy, where the medium position of the elements sharing the same suspiciousness value is used (average case).

Equation 2.5 shows the *absolute average rank* calculation [6], where i and f are code elements, the latter being the faulty one, while s_i and s_f are the respective suspiciousness score values.

$$E(f) = \frac{|\{i|s_i > s_f\}| + |\{i|s_i \geq s_f\}| + 1}{2} \tag{2.5}$$

Another issue arises when a program has multiple faults, which is common. Typically, the E value linked to the element with the highest suspiciousness score is used ($min(E(f))$, where $f \in \{\text{faulty elements}\}$). We will use this as the expense measurement in the following.

2.4 Results on DLFL

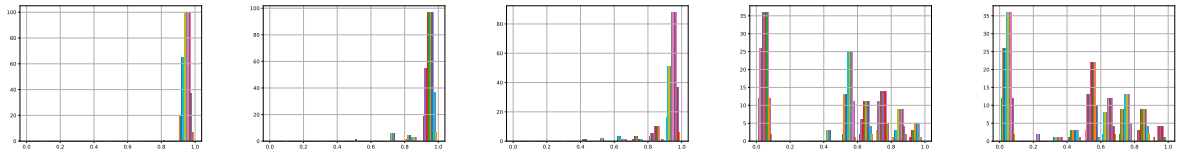
To examine the stability, various models were run on every selected program-versions 5 times in a row. The only difference between each 5 runs on the same input and model is the selected random seed, fixed at start. The separate runs have to produce same, or similar outputs, if the models are stable.

Table 2.6: *DLFL Average Expense results of 5 seperate runs of each version by different models*

	Mean			**Standard deviation**			**Minimum**			**Maximum**		
	MLP	CNN	RNN	MLP	CNN	RNN	MLP	CNN	RNN	MLP	CNN	RNN
Chart	419.54	460.50	521.47	312.28	225.47	232.54	124.81	223.69	293.00	853.17	760.52	830.00
Lang	201.89	329.80	306.34	122.63	179.35	139.67	82.19	134.65	162.25	385.27	569.00	493.98
Math	511.06	899.46	896.25	283.61	480.60	283.29	199.81	321.86	565.00	866.07	1454.44	1254.11
Mockito	522.48	707.43	679.70	310.54	402.02	295.12	184.43	259.93	315.62	906.22	1219.76	1038.05
Time	1618.37	1681.39	1422.41	571.25	941.65	506.16	871.86	638.21	912.07	2263.71	2857.71	2153.00
Average	**450.87**	**691.29**	**676.50**	**253.82**	**373.56**	**246.75**	**177.70**	**259.65**	**396.79**	**778.01**	**1145.81**	**993.17**

Table 2.6 shows a summary of results by programs and total. One can see in the first 3 columns the mean expense of the models on the benchmark, by programs. Columns 4–6 show the average of standard deviation, presenting the disparity from mean values of the five separate run by each version. These are quite high values, showing the high variety of the outputs of the separate runs, and it strengthens our conclusions of previous table. From columns 7–12 are the average maximum and minimum expense values of the five runs of models. As we can see there are really high differences between them, in most of the cases the maximal expense is 4–5 times bigger than the minimal one. The RNN model on Chart has proportionally the smallest distinctness, however that has still a double difference quotient. We also used statistical significance testing, by using Wilcoxon sign-rank test [26], complemented with Cliff's Delta effect size measure [59], which proved that maximum values are significantly larger than minimals, with large to medium magnitude of effect size in all cases. Detailed statistical test results appear in the online appendix [4].

Churn was measured in three settings: (1) by the original definition, (2) by aligning statements into boxes and (3) by filtering out boxes that do not contain faulty statements. In Figure 2.14, one can observe histograms of the measured churn values. These values were measured using the 5 trained MLP models, while other models show similar trends. Churn is measured on model variant-pairs, for example on version 1 between training attempt 1 and 2, next between training attempt 1 and 3, etc. These pairwise churn values are then averaged so that the histogram can more concisely represent all the measured data. It is clear from the figure that $Churn$ values are highest, $FlBoxChurn$ values the lowest and $BoxChurn$ somewhere between.



(a) *Origin churn* **(b)** *BoxChurn, box size = 5* **(c)** *BoxChurn, box size = 10* **(d)** *FlBoxChurn, box size = 5* **(e)** *FlBoxChurn, box size = 10*

Figure 2.14: *Histogram of churn values measured using the MLP model on the observed programs.*

It is application dependent what is an acceptable churn, but having an average value of 0.99 means that if the model is being trained again using different seeds, the statements will receive a different rank than before with 99% probability, which is clearly disadvantageous. One can argue that only the rank of the faulty statement is important, and that minor differences between ranks are still acceptable. To this end, we defined $FlBoxChurn$ and by measuring it on the subject programs we still see considerably high values. Notably, using a box size of 5, 65.08% of statements have a churn value higher than 0.5. For box size of 10 and 50, this is somewhat reduced to 64.94% and 60.94%, respectively. Average $BoxChurn$ value of box size 10 is 0.96, while for $FlBoxChurn$ it is 0.41.

In the 5 independent trainings performed, it was found that the output of the same model varies greatly due to the effect of random factors during training phase. Standard deviation of ranks is 291.37, with 606.22 mean values on average, implying 48.06% relative standard deviation, while based on boxed churn measurements the probability that two statements are going to end up in different boxes is 99.89%. These high values are good indicators of system instability.

2.4.1 Potential Improvement on Stability in DLFL

Metaparameter optimization techniques has been applied to fine-tune the three observed models, with the hope of achieving significant performance enhancements. However, despite exhaustive experimentation and resource allocation, the outcomes proved to be disappointing. On the other hand, model simplification proved effective. In Deep Learning it refers to the process of reducing the complexity and size of a neural network while maintaining acceptable performance levels. This technique aims to achieve several objectives, including improved model interpretability, reduced computational resource requirements, and enhanced generalization on limited data [12]. On the observed models the following architectural adjustments were made:

Table 2.7: *Effect of using the Resampling and the Simplified models combined on Average Expense results*

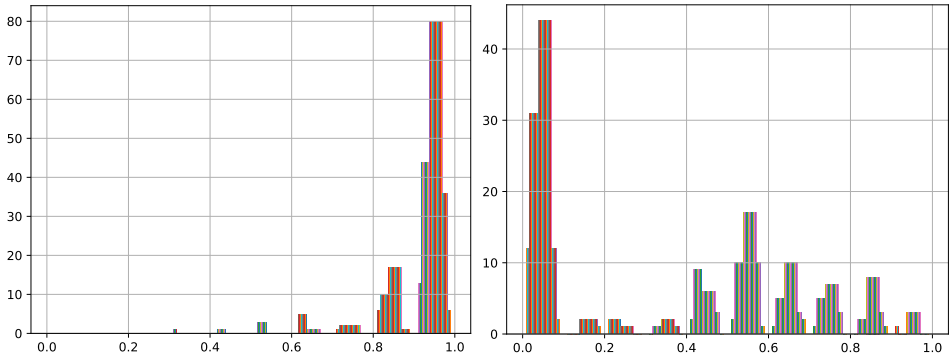
	Mean			Standard deviation			Minimum			Maximum		
	MLP	CNN	RNN	MLP	CNN	RNN	MLP	CNN	RNN	MLP	CNN	RNN
Chart	188.72	595.78	209.87	67.42	302.65	55.67	85.33	204.33	143.33	246.48	952.48	263.81
Lang	28.00	354.93	94.39	13.01	146.31	12.57	14.80	182.28	80.32	45.54	541.61	111.25
Math	124.05	860.38	209.13	51.32	432.69	58.57	67.88	361.98	150.30	193.82	1396.74	289.84
Mockito	336.86	617.54	278.98	117.34	298.77	103.02	193.19	270.32	146.16	468.57	985.00	404.30
Time	997.94	1945.97	337.43	413.92	1050.84	162.31	503.14	595.71	135.71	1514.43	3120.79	535.86
Total	163.64	687.35	191.91	63.62	337.15	55.61	87.88	289.17	128.78	241.11	1100.74	262.89
Diff	-287.23	-3.94	-484.58	-190.20	-36.41	-191.14	-89.82	29.51	-268.01	-536.89	-45.06	-730.28
Diff %	-63.71	-0.57	-71.63	-74.94	-9.75	-77.46	-50.55	11.37	-67.54	-69.01	-3.93	-73.53

- **MLP**: reduced the number of hidden layers
- **CNN**: the number of channels in convolution layers has been decreased to the 2/3 of the baseline
- **RNN**: the original two recurrent layers have been replaced by a single one

Resampling techniques can be roughly categorized into three commonly used types: oversampling, undersampling, and sampling with the creation of artificial data [200]. Oversampling was also applied to improve stability, which is simple yet effective and incurs a little cost. The approach first identifies failing test cases; then iteratively resamples failing test cases into original test cases; finally stops the iterative resampling process until obtaining a balanced test suite, where the number of failing test cases is the same as that of passing test cases.

In Table 2.7, we can see the mean values of DLFL using the two new improvement technique. The structure of table is same as for Table 2.6, but there are two extra rows, which show the absolute and percentage difference compared to the baseline. We can see that all models improved in average expense and in standard deviation as well. There are 70% improvements at MLP and RNN model in standard deviation, which indicates that stability seems to be improved. The maximal and minimal expense values both are highly decreased, however their ratios still seems to be too high to call models stable. Like in the previous section, CNN seems to be not really improved, the combined technique does not make it more stable. We used statistical significance testing here as well: the maximal values, despite the improvements, were still significantly greater than minimals in all cases, however the effect sizes decreased a bit in some cases.

Churn measurements support the statistical analysis. In the following discussion, experiments were carried out with a box size of 10. In Figure 2.15, histograms of boxed churn values are depicted using the MLP model. The histograms therefore represent the same data as in Figure 2.14, but the bars on the left side of the x-axis are



(a) *BoxChurn, box size = 10* **(b)** *FlBoxChurn, box size = 10*

Figure 2.15: *Histogram of churn values measured using the simplified MLP model and resampling.*

higher than before. This essentially means that churn has been somewhat reduced. To quantify this improvement: the churn value decreased by 0.0063 on average, resulting an average value of 0.98 in case of $Churn$, 0.93 in case of $BoxChurn$ and 0.32 in case of $FlBoxChurn$. Although the improvement is measurable, this can only be considered as moderate success.

Enhancement of stability and applicability can be achieved to a certain extent; however, the improvements may not suffice to ensure consistently reliable outcomes. Both statistical and churn measurements confirmed that stability improved, but the models remained insufficiently stable to produce reliable results in practical applications.

2.5 FixJS: Data Collection to Support APR

Lacking sufficient commit information, the evaluation of the proposed APR methods is always difficult. The aim of the dataset creation was to ease this burden by providing a dataset that can contribute to the efforts of the community. The result is the *FixJS* dataset, which is available on GitHub [4] and have a DOI to make it easily citable [5]. It contains roughly two million bug-fixing commits from GitHub.

2.5.1 Bug-fix mining

Two external tools were used in this phase: GH Archive [55] to retrieve commits from a specific time range and GitHub REST API [57] to get detailed information

[4] `https://github.com/RGAI-USZ/FixJS`
[5] 10.5281/zenodo.6340207

about a commit. To start off, we fetched every *push event* from GH Archive ranging between 01.01.2012 and 30.12.2012. Since GH Archive stores the commit hash and the commit message as well, bug-fixing commits were filtered out in this step. All commit messages containing one of the following keywords are identified as a bug-fix: `["fix", "solve", "bug", "issue", "problem", "error"]`. The same patterns are used in the work of Tufano *et al.* [168] and a similar approach in [49]. 2.129.715 bug-fixing commits were retrieved. These commits are saved in a csv file contating the date, event type, commit hash (sha), message and url in a monthly breakdown.

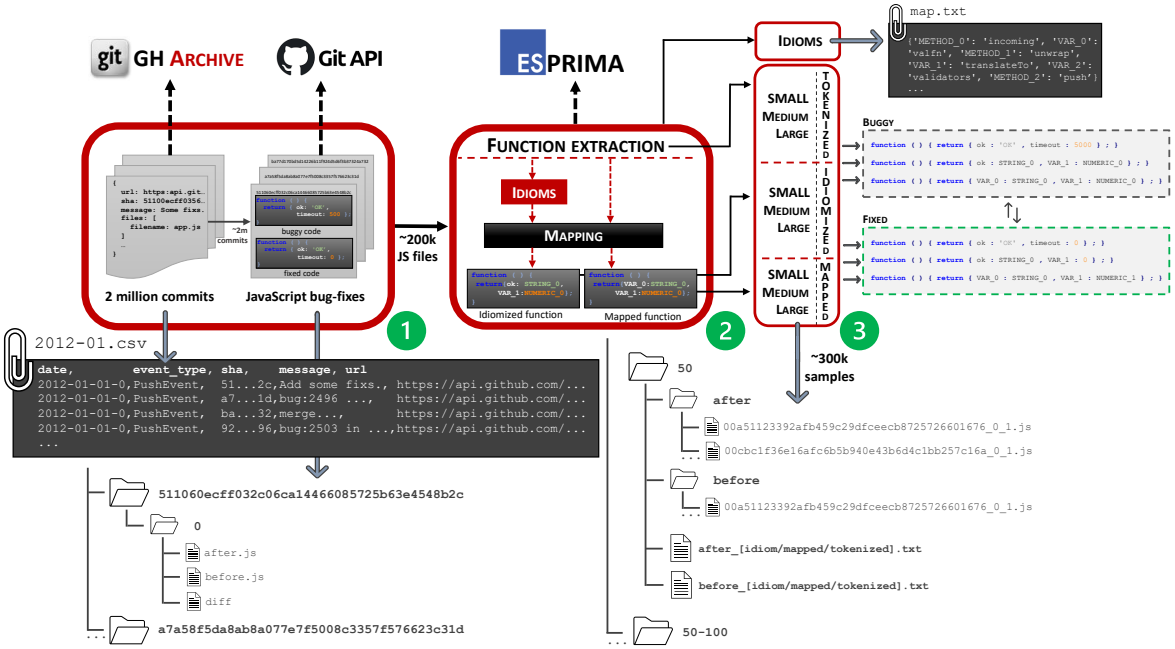


Figure 2.16: *A high level overview of the dataset creation approach*

Next, files are being fetched that are affected by the commit. Using GitHub API [57], non-JavaScript files were filtered out (files with not ".js" extension) and download the before- (i.e. buggy program) and after (i.e. fixed program) state of it. During the process some of the commits were ignored because their repository were renamed or deleted. At the end of this phase 103.115 commits were identified containing 201.198 files overall. These commits are saved in a folder named by their sha, containing three files: *before.js* (the JS file before the bug-fix) *after.js* (the JS file after the fix) and *diff* (the git diff of these files).

2.5.2 Patch Abstraction

The goal of this phase is to (1) identify the modified functions and (2) create a representation of it that can be fed into an AI model. Functions are extracted without their

names. Function expressions and arrow functions do not have names by definition in JavaScript, while function statement and member functions do have. The overwhelming majority of bug-fixes take place at the body of a function, thus ignoring names does not result with significant data loss.

Three different source code representations were createed for each raw function. To retrieve the Abstract Syntax Tree (AST) of the observed function, the Esprima library [45] was used. During this phase syntactical errors were filtered out. Part of the following representations are adapted from [168], while others are introduced before in Chapter 1.4, thus here their specification in full length is omitted. A small example on the right part of Figure 2.16 depicts this. Note that, in an AI-model point of view the main difference is in vocabulary size. While in the tokenized function every kind of identifier and literal can occur (resulting in a vocabulary of arbitrary size), in the other two representations the vocabulary size is fixed. We created three datasets with different token lengths: *small* (#tokens <50), *medium* (50 <= #tokens <100) and *large* (100 <= #tokens).

Tokenization

In this representation the function is split into tokens without any further modifications. Each token is separated by a space in the dataset. The vocabulary size is arbitrary.

Full-mapping

We call mapping the process in which identifiers and literals are mapped to generic IDs. Every ID follows the pattern `TYPE_INDEX`, where `TYPE` is the corresponding token type, while index ensures that each ID is unique in a before/after function pair. The indexing is sequential: when the parser finds e.g. an identifier, it will assign the ID `METHOD_0` to it, the second method will have the ID `METHOD_1`, and so on. The used types are the following: `[STRING, NUMERIC, BOOLEAN, REGULAREXPRESSION]` for literals and `[VAR, METHOD]` for identifiers. Vocabulary size <130 + I (where I is the sum of the largest index in each of the mapped keywords, 130 = 6 defined types + 63 JS keywords + ˜60 special characters).

Idiomization

Idiomization is the generalization of the full-mapping representation. Frequent identifiers and literals are often referred to as *idioms* [20]. In some cases they appear so often in the code that, they can almost be treated as keywords of the language (e.g. `i, j, 0, -1`). To retrieve these common keywords we counted the frequency of every token present in the fetched commits. From this then we picked the TOP-N idioms (N is arbitrary), let us call it the idiom-set. When parsing, and a token occurs,

the parser first examines whether it is present in the idiom-set. If yes, the token value is being used, otherwise the same mapping process is executed as in full-mapping. Vocabulary size <N + I + 130 (where N is the number of idioms, other same as before).

Table 2.8: *Summary of the constructed datasets.*

	# Tokens	# Samples	Size (mb)
Small	#tokens <50	67,070	78
Medium	50 <= #tokens <100	70,816	180
Large	100 <= #tokens	186,021	5,350
Overall		323,907	5,608

The token number (#tokens) determined using the Esprima [45] standard parser. Note that in #tokens each literal counts as one token. This can be confusing especially for string literals if they are not mapped (since they typically consist of multiple syllables).

2.5.3 Structure of the Constructed Dataset

FixJS distinguishes three source code representations in three different sized setting. Functions are separated based on token numbers and organized the files in different directories. Each of the folders contain seven text files: a mapping file, three before and three after files. In the latter mentioned files each line corresponds to a function of a specific representation. For example the sixth line in `after_idiom.txt` is the sixth bug-fix in 2012 that affects a JavaScript file and is preprocessed as described in Section 2.5.2, the corresponding buggy function can be found in `before_idiom.txt` in the sixth line. The same applies to the tokenized and mapped representations as well, the before/after state of the functions can be connected using their index in the files. The `map.txt` is the mapping file which contains the real world identifiers that are replaced in the tokenized and idiomized representations. Each line contains a dictionary where the keys are the IDs from the parsed function, while the values are the real world namings. In Table 2.9 the size of the assembled dataset is presented. Here we can see that the *Large* subset contains the majority of the samples, this and the sheer size of the functions implies that its size in megabytes is also the greatest. FixJS contains both single- and multi-line bugs. Before- and after state of the mined functions are differentiated using their Abstract Syntax Tree, meaning that if only comments have changed the samples are filtered out.

The dataset is available on GitHub, containing a detailed README of the featured files. To use the resulting dataset one should carry out the following steps:

1. Clone the repository and pick a dataset size (50, 100 or 100+)

Table 2.9: *Summary of the constructed datasets.*

	# Tokens	# Samples	Size (mb)
Small	#tokens <50	67,070	78
Medium	50 <= #tokens <100	70,816	180
Large	100 <= #tokens	186,021	5,350
Overall		323,907	5,608

The token number (#tokens) determined using the Esprima [45] standard parser. Note that in #tokens each literal counts as one token. This can be confusing especially for string literals if they are not mapped (since they typically consist of multiple syllables).

2. Load the before_*rep*.txt and after_*rep*.txt (where *rep* can be [idiom, mapped, tokenized])
3. Split the dataset (*e.g.* 80-10-10) and train the model
4. Evaluate the model on the test set

The possible uses of FixJS is quite generic and similar to existing datasets like Defects4J or BugsJS. However, these databases are small in size to teach a deep learning model, and their preparation requires a serious development effort. On the other hand, the bug-fixes in FixJS are already extracted and organized in quite large quantities. Real world bug-fixing commit information facilitates automatical software refactoring and may improve software evolution. It can enable seamless integration between continuous code changes and serve as a ground to better understand the software development cycle. The proposed dataset serves these goals, it can provide a common ground in evaluating data-driven repair solutions, potentially contributing to a better understanding of the strengths and weaknesses of different source code extraction methods and lead to their best combination. It provides more detailed data than the currently available alternatives and can also be used in different representation evaluations.

FixJS includes ∼300k samples containing separately the buggy and fixed codes. It comes in three sizes: small, medium and large. In each of these datasets three source code representations with different abstraction levels are present. The dataset is mainly intended for Automated Program Repair research evaluation purposes.

2.6 APR with a pre-trained model

To automatically generate patches, experiments were conducted using ChatGPT [133]. The original Generative Pre-trained Transformer, or GPT for short, was published in

2018. ChatGPT is a descendant of this architecture. Its base is a Transformer, which is an attention model that learns to focus attention on the previous words that are most relevant to the task at hand: predicting the next word in the sentence. ChatGPT is fine-tuned from a model in the GPT-3.5 series, which finished training on a blend of text and code in early 2022. At the time of writing this paper, some details of the underlying architecture of ChatGPT are unknown, but the research community knows that ChatGPT was fine-tuned using supervised learning as well as reinforcement learning [22]. In both cases, humans were involved to improve the model's performance by ranking answers from previous conversations and imitating conversations [133]. Although GPT-4 became available while writing this thesis, in my experiments, I used the GPT-3.5 version of ChatGPT.

2.6.1 Prompts to generate patches

At the time of writing this thesis (Q2 2024), ChatGPT is available via API and also via a graphical interface, where users can communicate with the model by inputting a prompt [134]. To fix a candidate buggy function, I experimented with different configurations: the input of the model consists of the sample code snippet from the observed datasets + one of the below-listed prompts. These prompts are proposed by myself, but note that the choice of these is arbitrary, and I included the below ones because during the experiments, I found them interesting. The following prompts are proposed:

P_1: `Fix or improve the following code:` *[code]*

The most natural way to prompt the model is to just input the buggy function with the instruction to fix it.

P_2: `Modify the following Java/JavaScript code:` *[code]*

Based on our observations, the keyword *fix* or *repair* can confuse the model: it looks for a syntactical error, but in most cases, the bug is semantic - thus only refinement/-modification is needed.

P_3: `Fill in the missing part in the following:` *[code]* `___` *[code]*

By deleting the original buggy part of the code, we force the model to generate something in its place.

P_4: `Continue the implementation of the following function using X tokens:` *[code]*

Since the underlying model (GPT-3) was trained to estimate the next word in a sequence, it makes sense to use the first few statements in the code and ask ChatGPT to generate the rest.

P_5: `Fix or improve the following code (with bug location hint):` *[code]* `* Refinement starts here *` *[code]* `* Refinement ends here *` *[code]*

Essentially the same as the first prompt listed here, but here the bug location is marked with comment blocks. The specification of the exact location of a bug might

not be too realistic since real-life bug localization is usually less precise, but for the sake of experimentation, it might provide interesting insights.

2.6.2 Evaluation of the generated patches

Since the goal of ChatGPT is to mimic a human conversationalist, it is in its nature that answers are long and explanatory, with a lot of natural language text. Thus, the use of standard evaluation metrics (e.g., precision, recall) is not possible. Therefore, I manually analyzed the answers and classified them into one of the following categories:

1. **Undecided**: when we were uncertain about the correctness of the response or ChatGPT was unable to generate a fix
2. **Incorrect patch**: the output code is different from the developer patch
3. **Fix in answer**: the generated answer contains the correct fix for the given bug
4. **Semantical match**: the proposed fix semantically matches the one generated by a human engineer
5. **Syntactical match**: the returned fix is the same as the developer patch, except for whitespaces

2.6.3 Repair performance of ChatGPT

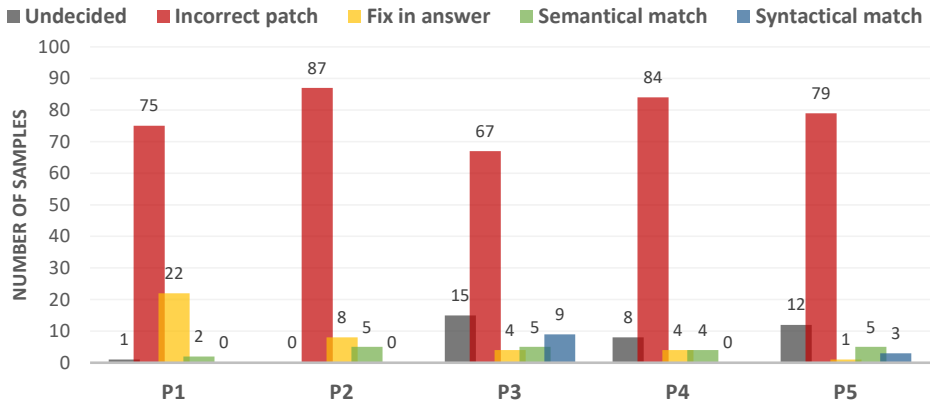


Figure 2.17: *Manually evaluated results of ChatGPT on the Java dataset*

In this section, results are presented obtained by feeding 100 Java and 100 JavaScript samples to ChatGPT using 5 different prompts (forming a total of 1000 trials/input-output pairs). Figure 2.17 and Figure 2.18 show the manually evaluated results for

Java and JavaScript, respectively. Upon examining the figures, it is evident that ChatGPT is not proficient in generating patches that semantically or syntactically match the developer fix. Instead, it provides suggestions on how to repair the code or generates a fix that contains the correct solution. The manual evaluation also revealed that different prompts trigger different response mechanisms from the language model. For instance, candidates generated using P_2 often involve significant code changes, rarely deleting or simplifying code snippets, but rather creating more advanced solutions. Another observed pattern is that, for the Java dataset and P_5 , the generated answers typically contain a `try-catch` block (which is less apparent in the case of JavaScript).

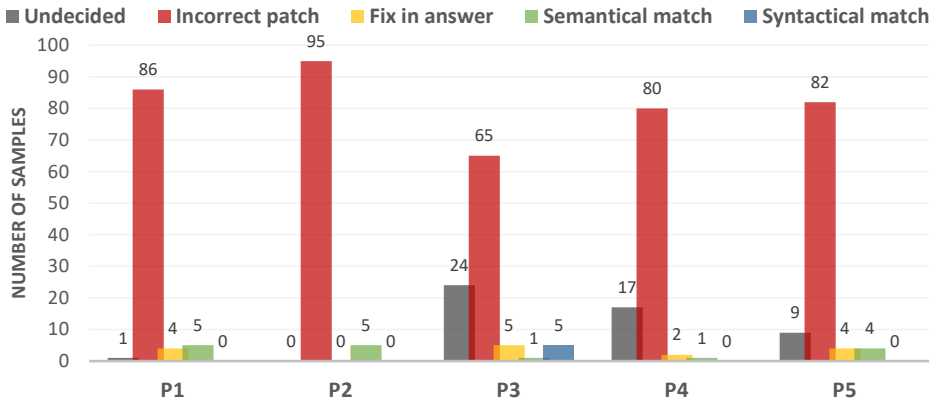


Figure 2.18: *Manually evaluated results of ChatGPT on the JavaScript dataset*

Based on the results, it can be concluded that prompts have a major effect on the repair performance of ChatGPT. Among the proposed prompts, best results have been achieved in terms of repair suggestions using P_1 and P_3 in terms of syntactical matches.

The answers generated using prompts P_3 and P_5 often did not include the fix because ChatGPT was unable to generate it. This phenomenon can be observed in cases where significant changes were made during the bug fix, and these prompts essentially delete the modified part, leaving the model with little information about the code's purpose. We also noticed that even small modifications to a prompt can have a significant impact. For example, modifying P_3 to include the number of tokens to be generated (i.e.: `Fill in the missing part in the following code applying X tokens:`)resulted in a significant performance drop. The answers consistently included combinations or reformulated versions of statements such as (1) the missing part cannot be filled with the information provided, (2) in order to complete the code, more context and information about the specific implementation is needed and (3) the information provided is not sufficient for me to understand the context

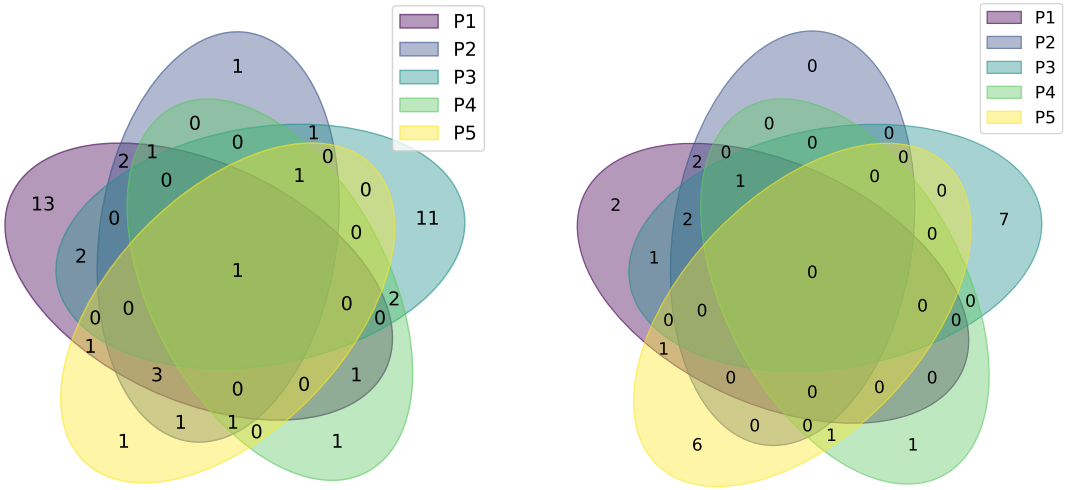
and purpose of the method. The prompt P_5 also often resulted in "undecided" answers. However, here it can be attributed to the fact that the original code is usually syntactically correct in its context, and the bug fix usually only makes sense when observing a larger context or when it is a simple code refinement. Thus, the semantics of the code remain the same.

An interesting insight is that different prompts tend to modify the code in different styles, which suggests the possibility of classifying which prompt fixes which types of bugs, thereby optimizing the repair performance. We observed similar patterns in both Java and JavaScript cases. However, as shown in Figure 2.17 and Figure 2.18, ChatGPT suggested significantly more correct fixes for Java. Based on our empirical observations, we hypothesize that this difference is due to the fact that JavaScript developers often use custom object creations (e.g., `{key: val, ...}`) and diverse libraries, while Java follows more standardized conventions with commonly used keywords and methods. Overall, we can conclude that the two languages are distinct and differ greatly in design. Additionally, in the JavaScript dataset, function names are often omitted due to anonymous and arrow functions, whereas in Java, function names are present, which helps the model in understanding the purpose of the function.

Although the possibility of dataset bias cannot be excluded, based on the empirical evaluation, ChatGPT tends to generate better repair candidates for Java and less satisfactory ones for JavaScript. Further research in this area is required to determine the exact reasons for this difference in performance.

One might wonder why variable names are not generalized in the used samples, as it is a common practice even in state-of-the-art approaches to reduce vocabulary size [111]. Without in-depth analysis, we experimented with placeholders but experienced a decrease in performance. It seems that actual variable names and types are beneficial in bug-fixing with ChatGPT. Without them, the answers usually included the following observations: (1) TYPE_1 and TYPE_2 are not defined as actual types, and you will need to replace these placeholders with the appropriate types, or (2) METHOD_1 is not implemented, and you will need to provide the implementation. Overall, it appears that language models, such as ChatGPT, contain the most common names and types, so masking them is not beneficial.

Different prompts tend to generate independent fix templates, and this behavior is also observed for Java and JavaScript. For example, P_2 always modified large chunks of code, while P_1 modified only some parts or even left it untouched. Based on these observations, the results in the Venn diagram in Figure 2.19 are not surprising. The diagram illustrates the distribution of correct fix answers among the used prompts, representing the overlap of fixed bugs using the proposed prompts. In the case of Java, there is only one sample that was fixed by all prompts (a variable name



(a) *Java - overall 44 distinct fix* **(b)** *JavaScript - overall 24 distinct fix*

Figure 2.19: *Distribution of correct fix answers in the used prompts*

change was necessary, see example 50 in the online appendix), while in the case of JavaScript, there were none. Furthermore, in the Java dataset, ChatGPT repaired 44 different bugs (including answers with the correct fix, semantically identical patches, or syntactically identical ones), while in the JavaScript dataset, it repaired only 24. This also demonstrates that prompts trigger different repair mechanisms, and choosing the right one is a crucial decision.

> *Since there is insignificant overlap in the fixed bugs and different prompts tend to repair different types of bugs, no particularly easy-to-fix bugs have been found during the experiments. However, some bugs are more likely to be correctly patched by a well-chosen prompt rather than a poorly chosen one.*

Discussion

Overall, results show that choosing the right prompt is a key aspect as it biases the generated fix in many ways. The impact of programming languages is not negligible. It seems that LLMs, such as ChatGPT, tend to generate higher-quality patches for Java (a classic OOP language) and lower-quality patches for JavaScript (an event-driven, functional language). One possible explanation might be that Java is more human-like and its readability is more natural compared to JavaScript. However, to understand the in-depth consequences, further research in the area is needed.

Although the same patterns are apparent in the results, the overlap of fixed bugs is insignificant, and we did not find easy-to-fix bug types in either of the datasets. It is clear that choosing the right prompt holds great importance. On average, Chat-

GPT was able to propose corrections in approximately 19% of cases, but choosing the wrong input format can drop the performance to as low as 6%. Compared to Transformer models that were fine-tuned to automatically repair programs, this accuracy is not significantly high. For example, on the CodeXGLUE benchmark [111], the highest-ranking approach achieved 24% on the small dataset, which is comparable to our data. On the other hand, ChatGPT is surprisingly effective at generating repair candidates, despite not being explicitly trained for that purpose.

As language models continue to improve and evolve, it is possible that prompts that were effective in the past may become less effective or even irrelevant in the future. One approach that can be used to mitigate the impact of model evolution is to periodically re-evaluate the model and prompt selection process to ensure their continued effectiveness.

2.7 Genetic Automated Program Repair

Since the appearance of GenProg [174] and its genetic approach, many excellent researchers tried to improve the performance of it by creating several distinct APR tools [63, 114, 116, 119, 145, 169, 190]. Many of these follow the Generate-and-Validate approach [58], that is, first a patch is generated and than the test suite is executed to check the correctness of the generated candidate. In this section GenProgJS is presented, a general automated repair tool which generates repairs for real-world JavaScript bugs. To the best of my knowledge it is the first APR tool, which uses genetic algorithm to generate patches for JS. It is built in a way, that the progress of patch creation is well traceable, making it possible to further analyse the usefulness of operators and meta-parameters. This tool is an implementation of GenProg [98] for JavaScript, with greater or lesser language-specific improvements.

The source code of each individual in the population is represented as an AST, which holds additional information to make the required code transformations. Although JavaScript (partially) supports Object Oriented Programming (OOP), it is not guaranteed that every source file in a project is written with this in mind. Thus, unlike in Java, where the search space for repair code snippets is usually class level (or package level), in case of JavaScript the fixes were extracted from the bugs own source file.

2.7.1 GenProg for JavaScript

GenProgJS adapts the genetic approach of GenProg, adding some new operators, to boost its performance on JavaScript. In this section first the applied genetic algorithm is presented, then the used mutation and crossover operators.

Genetic Algorithm

The implementation follows the generic schema of a genetic algorithm, as can be seen in Algorithm 1. The process starts with the initialization of the population, which basically means to load the buggy program into the memory. The algorithm runs until it reaches the maximum number of generations, which is an input parameter. The essence of the algorithm starts at line 4: the (re)calculation of the fitness values is carried out here. The calculation goes along a predefined formula which can be seen in Equation 2.6, where *P* denotes a program variant,W_{pos} and W_{neg} are the weights for positive (passed) and negative (failed) tests, respectively. In the equation $T_{pos}(P)$ and $T_{neg}(P)$ denotes the set of passed and failed tests respectively on a *P* program.

$$fit(P) = W_{pos} \times |\{t|t \in T_{pos}(P)\}| + W_{neg} \times |\{t|t \in T_{neg}(P)\}| \tag{2.6}$$

After updating the fitness values, the *offsprings* set is being initialized, in which the descendants of the given population are stored. At the beginning of the iteration a random number is generated, which reinforces the presence of randomness, as we can see at line 9 in the algorithm. As the iteration goes through all the mutation operators, the probability of each operator is examined – this value can be specified in creation time, and if it is greater than the generated random number, the operator gets the chance to operate on the given individual. Thus, if an operator is applied to the current population, new individuals are added to the set containing the descendants. Once this is done for all the operators, the next important statement can be found at line 14. The situation is similar to the previous operators, but here a descendant is obtained by crossing two parent elements and not mutating only one individual. After that a new population is selected from the generated offsprings using genetic selection.

Selection makes use of the fitness values: those individuals stay in the population, which have the smallest fitness values, as many as the maximum population number (an adjustable parameter). To give space to randomness, first an *n* times larger set is selected than the size of the population, and after this the individuals are selected randomly. At the end the iteration counter is increased and the set of potentially improved programs are expanded. While it is true that at the end of the iteration, in principle, the population contains only improved individuals, the size of the population is limited, and potentially improved programs may be lost, not to mention the side effects of randomness. A ”theoretical guarantee” cannot be given that the generated program variants are correct, but one can have a guess by checking the fitness value of a given variant – based on this it can be determined if it does not contain failed tests.

Algorithm 1 The high-level algorithm of the GenProgJS approach

Input:
- P buggy program
- a set of passed T_{pos} and failed T_{neg} tests
- *MG* max generations
- *OPS* set of mutation operators

Output:
- *REP* set of potentially fixed program variants

Algorithm:

1: $popul_0 \leftarrow init\_popul(P)$
2: $i \leftarrow 0$
3: $REP \leftarrow \emptyset$
4: **while** $i < MG$ **do**
5: $refresh\_fitness(popul_i, T_{pos}, T_{neg})$
6: $offsprings \leftarrow \emptyset$
7: **for all** $o \in OPS$ **do**
8: $r \leftarrow rand()$
9: **if** $r < o.probability$ **then**
10: $offsprings \leftarrow offsprings \cup op.operate(popul_i)$
11: **end if**
12: **end for**
13: $r \leftarrow rand()$
14: **if** $r < crossover.probability$ **then**
15: $r_1, r_2 \leftarrow rand()$
16: $offsprings \leftarrow offsprings \cup crossover(popul_{r_1}, popul_{r_2})$
17: **end if**
18: $popul_{i+1} \leftarrow select(offsprings)$
19: $i \leftarrow i + 1$
20: $REP \leftarrow REP \cup \{o \mid o \in offsprings \ \wedge \ o.fitness == W_{pos} \times |T_{pos}|\}$
21: **end while**

Operators

Astor [119] is an automated software repair framework for Java. Its purpose is manifold, in our case we adapted some of the mutation operators from their implementation to GenProg. GenProgJS also contains JavaScript operators, which are not present in other repair tools. These operators have been derived by examining works about the most common JavaScript errors [65, 131, 135]. Hanam *et al.* [65] categorized JS bugs in six big groups, based on a large scale experiment on open source projects. According to them, these categories are: (1) Dereferenced Non-Values, (2) Incorrect Comparison, (3) Missing argument, (4) Incorrect API config, (5) `this` not correctly bound, and (6) Unhandled exception. By manually inspecting the BugsJS dataset we found that these categories rather well specify the erroneous behaviour of JavaScript. With these in mind mutation operators have been designed, which try to tackle with these challenges. In addition operators which create patches that span through multiple lines (i.e. multi-line repairs) are also included in the tool. Since the implementation of these operators is not direct work of the author, their description is opted out from the dissertation but can be found in the online appendix of the tool [53]. Unlike mutation operators, *crossover operator* operates on two program variants, which were selected beforehand by the Python API. Basically it works as follows:

1. The operator gets the AST representation of each parent individual, where the buggy lines are.
2. Selects a subtree from both of them.
3. Replaces the subtrees with one another.
4. Selects one of the offsprings.

This offspring then passed back to the genetic algorithm, which handles it as a new individual. We designed the implementation to be as generic as possible: the described approach is `TreeCrossover`, which extends the base `Crossover`, thus allowing future extension (*e.g.* crossover operating on raw source code instead of AST).

2.7.2 Dataset and experiment setup

The BugsJS dataset [64] contains reproducible JavaScript bugs from 10 open-source Github projects. The dataset contains both single- and multi-line bugs as well. These projects are listed in Table 2.10. It's not listed in there that in the original dataset there are **453** bugs, but not every one of them are ”repairable”, because of the lack of failed test cases. Thus, the number of bugs, for which there is at least 1 failed test is **352**. For the very same reason, 3 projects which are present in BugsJS are exluded

from our experiments, namely Hessian.js, Shields and NodeRedis. BugsJS features a rich interface for accessing the faulty and fixed versions of the programs and executing the corresponding test cases. These features proved to be rather useful for the comparison of automatically generated patches with the ones which was created by developers. GenProgJS also incorporates static data from BugsJS, which includes bug location indices as well, meaning that the implemented tool currently does not include bug localization, its main purpose is to generate patches automatically.

Table 2.10: *Systems contained by the BugsJS dataset.*

System	kLOC	#bugs	Github URL
Bower	16	3	https://github.com/bower/bower
Eslint	240	290	https://github.com/eslint/eslint
Express	11	27	https://github.com/expressjs/express
Hexo	17	7	https://github.com/hexojs/hexo
Karma	12	6	https://github.com/karma-runner/karma
Mongoose	65	12	https://github.com/Automattic/mongoose
Pencilblue	16	7	https://github.com/pencilblue/pencilblue
Total	377	352	

The GenProgJS tool comes with a number of adjustable parameters. Based on the literature the size of the population should be kept relatively low, thus we chose it to be 20 in our experiments. Keeping the number of generations low is implied in some way by the previous number, since for a few individual many operators can be applied relatively quickly. One can choose from five stopping criteria when using the GenProgJS tool: (1) stopping by reaching the maximum generation number, (2) running time goes beyond the specified value, (3) reaching the specified number of repair candidates, (4) all of the above criteria must meet, or (5) any of the above criteria must meet. In the experiments the algorithm was terminated when it reached the maximum number of generations of 30, but this has a relatively small significance, since a potentially repaired program is usually created in earlier generations. Accordingly, it is dependent on the complexity of the repair, and in other scenarios alternative stopping criteria may be more useful. The weights in the fitness function are based on the original GenProg implementation: the value of W_{pos} and W_{neg} are 1 and 10, respectively. Although other fitness functions are also used [119] ranging from simple ones such as the number of failed tests to rather complex ones, the point of them is the same: to punish negative test execution in some way with some weight. Last but not least, the probability of using each operator must be set. Because the benefits of using each operator are very difficult to quantify, these probabilities have not been optimized. The probability to use an operator is 0.8 for every operator. All experiments were executed on a computer with an Intel(R) Xeon(R) CPU E5-

2630 v4 @ 2.20GHz processor. The random nature of the genetic algorithm makes it harder to validate and reproduce our results, thus, for the sake of generalization our experiments were carried out on 5 independent runs.

2.7.3 Results and Discussion

In this section, the featured automated program repair tool is evaluated and discussed. First, the overall repair performance is summarized and then further examination of some repaired bugs is presented to provide analysis of the use of repair operators and the searching process.

Repaired bugs

GenProgJS found plausible patches for **31** bugs in the BugsJS dataset. In Table 2.11 basic data is collected of these repairs. In order to save space, the following abbreviations are being used: F-Feature, Impl-Implementation, Expr-Expression, Decl-Declaration. The shapes in the fifth column have the following meaning: ✓✓ - syntactically identical, ✓ - semantically identical, × - uncertain The fourth column (*Average runtime*) averages the time it took to find the first plausible patch over the five independent runs. For each bug the BugsJS Dissection site[6] provides several useful information on its nature and the bugfix type it requires. In the table, next to the Bug Id, the bug type according to the dissection taxonomy is shown. The 31 repaired bugs cover a variety of bug types (**31** bugs belong to **14** types of bugs) and belong to **6** different open-source projects, which shows that the system indeed provides a general way of bugfixing. Although it is true that almost half of the fixes were generated for the Eslint project, worth noting that in the BugsJS database there are way more bugs for this project than for others (see Table 2.10). The overall 31 bug repairs is a good starting point for JavaScript program repair, GenProg implementations for C (10) and Java (9) started with even less bug fixes [116, 174]. Considering that state-of-the-art repair tools can repair up to 30 bugs from a well-known database does not make these more efficient, since most of the fixed bugs are non-exclusives [97]. Also, many authors in the APR community reason that the number of patched bugs by each repair tool is dependent of famous bug databases (e.g. Defects4J) [43] and their precision/performance could dramatically drop on other previously unseen databases.

The search for plausible patches was not stopped at the first one that is found, each experiment went until reaching 30 generations. This resulted a variable number of plausible patches in each run. We experimented with two different settings of the tool, which we call (1) *generator* and (2) *mutator*. In the first case, when the tool founds a plausible patch, stores it and removes it from the population of program variables. This approach encourages the tool to generate different patch

[6] https://bugsjs.github.io/dissection/

Table 2.11: *Repairs on the BugsJS dataset produced by GenProgJS.*

	Bug Id	Taxonomy	Fix type	Average runtime	Identical found?	# run (Plausible patches) 1	2	3	4	5
1.	Bower 2	Incomplete F. Imp. → Configuration Processing	Chg. of If Condition Expr.	87646	✗	-	2	-	-	1
2.	Eslint 1	Incomplete F. Imp. → Missing Input Validation	Chg. of If Condition Expr.	15988	✗	1	3	4	6	-
3.	Eslint 7	Incomplete F. Imp. → Missing Input Validation	Multiple	51	✗	15	16	-	17	29
4.	Eslint 41	Incomplete F. Impl. → Missing Input Validation	Chg. of a return statement	35899	✗	-	2	3	-	1
5.	Eslint 47	< uncategorized >	Chg. of Class Field Decl.	113	✓✓	1	1	3	1	1
6.	Eslint 72	Incorrect F. Impl. → Incorrect Output Message	Chg. of Class Field Decl.	1141	✓✓	3	4	7	5	7
7.	Eslint 94	Incorrect F. Impl. → Missing Type Check	Chg. of If Condition Expr.	357	✗	70	94	14	67	114
8.	Eslint 100	Incorrect F. Impl. → Missing Type Check	Chg. of Assignment Expr.	1447	✓	20	26	12	9	26
9.	Eslint 122	Incorrect F. Impl. → Incorrect Input Validation	Chg. of Method Decl.	219	✗	1	-	-	1	-
10.	Eslint 130	Incorrect F. Impl. → Incorrect Data Processing	Chg. of Assignment Expr.	8293	✗	-	-	1	-	-
11.	Eslint 154	< uncategorized >	Chg. of Assignment Expr.	2155	✗	21	-	25	-	29
12.	Eslint 158	Incorrect F. Impl. → Incorrect Input Validation	Chg. of Method Call	481	✗	-	-	34	24	20
13.	Eslint 217	Generic → Typo	Chg. of If Condition Expr.	1329	✓	6	5	4	4	6
14.	Eslint 221	Incorrect F. Impl. → Incorrect Input Validation	Chg. of a return statement	278	✓	100	334	221	202	183
15.	Eslint 321	Generic → Typo	Chg. of If Condition Expr.	12058	✓	9	4	5	8	13
16.	Eslint 323	Incorrect F. Impl. → Unnecessary Type Check	Chg. of If Condition Expr.	716	✓✓	260	192	198	159	220
17.	Express 2	Incomplete F. Impl. → Incomplete Data Processing	Chg. of Method Call	2333	✓✓	2	5	4	115	19
18.	Express 3	Incomplete F. Impl. → Incomplete Initialization	Chg. of Class Field Decl.	9961	✗	1	-	-	-	3
19.	Express 5	Incomplete F. Impl. → Incorrect Filepath	Chg. of If Condition Expr.	456	✗	23	61	57	15	100
20.	Express 8	Incorrect F. Impl. → Empty Input Parameters	Chg. of If Condition Expr.	357	✓	487	350	402	284	468
21.	Express 9	< uncategorized >	Chg. of Assignment Expr.	3428	✗	-	-	-	1	-
22.	Express 16	Incorrect F. Impl. → Incorrect Data Processing	Multiple	280	✗	20	-	20	20	19
23.	Express 18	Incorrect F. Impl. → Incorrect Data Processing	Chg. of If Condition Expr.	155	✗	122	151	67	110	127
24.	Express 26	Incomplete F. Impl.→ Handling Special Characters	Chg. of Assignment Expr.	8635	✗	-	1	-	-	-
25.	Karma 3	Generic → Missing Return Statement	< uncategorized >	9154	✓	1	2	3	2	-
26.	Karma 4	Incomplete F. Impl. → Missing Input Validation	Chg. of If Condition Expr.	7917	✗	-	-	1	-	-
27.	Karma 9	Incorrect F. Impl. → Incorrect Filepath	Chg. of a return statement	232	✗	28	30	22	20	31
28.	Mongoose 3	Incorrect F. Impl. → Incorrect Data Processing	Chg. of If Condition Expr.	1071	✓✓	166	286	226	516	270
29.	Mongoose 8	Generic → Missing Return Statement	< uncategorized >	59144	✓	-	4	9	22	12
30.	Mongoose 11	Incomplete F. Impl. → Missing Input Validation	Chg. of If Condition Expr.	8035	✗	300	477	382	588	147
31.	Pencilblue 4	Incomplete F. Impl. → Missing Type Check	Multiple	473	✗	5	3	4	4	-

variants. On the other hand, mutator does not remove the plausible patch from the population, but checks whether the patch was already generated (thus ensuring that every plausible patch is unique). In the latter case we observed that after the first plausible patch the process most likely re-used the plausible ones and from that point constantly generated new plausible candidates. Hence, the first plausible patch has an emphasized role in the process. Although we expected that the two settings will produce very different results, the outputs are essentially the same. The following experiments were carried out using the generator approach, although mutator would not give very different results.

In Table 2.11 next to the Fix type, the average repair time of the first repair candidate is listed in seconds. Despite that the algorithm was not stopped when it found a plausible patch, the importance of the first found patch is undeniable. Although it cannot be stated in general that only those operators are useful, which were in-

volved in the repair process of the first plausible patch, as we progress through the generations, more and more operators are employed and it is becoming increasingly difficult to decide which operator application was useful and which was not. We have also indicated in the table whether there is a plausible patch which is identical (or lies close) to the human patch.

As can be seen on Table 2.11, in **12** cases the generated patch semantically matches the developer fix. If one considers every patch, 7 out of 12 repair process involved a JS operator (58%), while if we examine only the first candidate, JS operators has only been used 3 times(25%). Interestingly there are cases where almost every applied operator was JS specific (e.g. Eslint 221 or Eslint 321), while no JS operator were involved during the synthesis of the first patch. During the repair process of Eslint 221, 5 JS operators were used, whereas for Eslint 321 only 1, which is *NullCheck*. This operator alongside with *VarChanger* were actually used in 6 out of 7 cases of the observed bugs. From this and from other empirical observations we would like to emphasize the prominent role of these operators in JavaScript program repair. They are applicable in a lot of real world scenarios and as we have seen here, it in fact creates correct patches. The *NullCheck* operator is more interesting in this sense, since it captures the dynamic nature of JavaScript and summarizes well the root cause of the misbehavior. Although *NullCheck* is an eminent operator it is worth noting that if we consider only the first patch, it was involved only in 1 repair process, while *VarChanger* in 3 processes.

To put our results in a context, we observed 18 test-suite-based state-of-the-art repair tools. Without claiming this to be the complete list of such tools, we did our best to place the most recent works on the list. Worth noting that data-driven approaches are intentionally excluded, since the evaluation criteria of those often differs from test-based repair. This exploratory research is based on the Living Review on Automated Program Repair by Martin Monperrus [128]. The found tools and their corresponding ”repair performance” can be found on Table 2.12. Precision means the precision of correctly fixed bugs in bugs fixed by each APR tool (x/y). Fix rate is the percentage of bugs for which the repair tool has generated a plausible patch ($y/\#bugs$). To the best of our knowledge GenProgJS is the first test-suite-based automated repair tool for JavaScript (and the first which was evaluated on the BugsJS dataset), we could not directly compare it to other approaches. On the other hand one can see that both the Fix rate and the Precision is comparable with other state-of-the-art repair tools. Note that the implementation of GenProg both for C and Java generated less correct-patches than for JavaScript. This result has little to do with the quality of the implementation, rather it is due to the unique characteristics of the observed datasets and/or the applied genetic operators. APR tools written for Java are usually evaluated Defects4J [81] and do not consider other datasets. That is why current tools can easily have the problem of overfitting: they are biased

Table 2.12: *Results of test-suite-based program repair tools.*

Repair tool	# bugs	x/y	Fix rate	Precision
GenProgJS	352	12/31	8.8%	38.7%
GenProg [174]	105	1/10	9.5%	10.0%
JGenProg [116]	224	5/27	12.1%	18.5%
JKali [116]	224	1/22	9.8%	4.5%
MutRepair [38]	463	-/25	18.5%	-
jMutRepair [117]	224	3/17	7.6%	17.7%
Nopol [190]	224	5/35	15.6%	14.2%
ARJA [198]	224	18/59	26.3%	30.5%
CapGen [118]	224	21/25	11.2%	84.0%
SimFix [79]	357	34/56	15.6%	60.1%
HDRepair [95]	90	-/23	25.6%	-
ACS [188]	224	17/23	10.3%	73.9%
ssFix [185]	357	20/60	16.8%	30.0%
ELIXIR [153]	82	26/41	50%	63.4%
JAID [23]	138	25/31	22.5%	80.6%
SketchFix [73]	357	19/26	7.3%	73.1%
FixMiner [92]	357	26/32	8.9%	81.3%
Prophet [109]	69	14/20	28.9%	70.0%
Average	231	14/31	17.5%	44%
Median	224	17/27	13.9%	38.7%

The column next to the tool name corresponds the total number of bugs on which the given approach was evaluated. In each row, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a plausible patch is generated by the APR tool (i.e., a patch that makes the program pass all test cases).

to the database. Although the available datasets are limited to Java and have been criticized recently [108], it is relatively well tested and mature compared to what is available for JavaScript.

In average it took the algorithm quite a long time to execute the 30 generations. The runtime depends heavily on the test suite: for instance the total execution time of the Eslint 72 bug was 33 hours, while for Karma 4 only 9 hours. Of course if we are only interested in the first plausible patch and stop the algorithm then, the execution time drops heavily. Also, running only the failed test cases in the algorithm is highly recommended. While the running time is far from optimal, so far, efficiency is not a widely-valued performance target in the field of APR. Authors in [97] found, that state-of-the-art APR tools are the least efficient. While execution time is an important aspect of a software artifact, it always depends on the hardware architecture. The simplest way to boost the speed of the algorithm, is to put more powerful hardware behind it. However, there are other options as well. One, so to speak, trivial speed up possibility is to run the test suite in a smarter way, meaning while the patches

are generated, first try to fit only on the failed test cases and later check whether the code transformation ruined the previously positive ones. This approach could result in a significant improvement, especially in projects where there are thousands of test cases. The test execution described above is not yet supported by GenProgJS. We made steps to make it available in the future, though we encountered technical difficulties (*i.e.* using Mocha –grep with inline tests).

In the following in the source code listing the original buggy line is marked with red background and - sign, the automatically generated patch with green background and + sign, while the developer fix with yellow background and > sign.

Repairing on script language

JavaScript does not compile and is weakly-typed, thus tolerates partially incorrect programs to run. This removes the part of the candidate validation typically offered by the compiler for other languages such as C and Java. This lack of validation on the one hand prolongs the process of patch synthesis (since the patches only rely on test executions), on the other hand, the algorithm is able to generate a whole new class of plausible patches, which are fundamentally different from what we get used to in traditional languages.

Table 2.13: *Bugs and their corresponding plausible patches.*

Bug Id	Operator	Original line	Plausible patch
Eslint 1	StringCut	`if (name === "Math" || name === "JSON" )`	`if (name <= "Math" || name <= "S" )`
Eslint 130	CallChanger	`notEmpty.forEach(function( x, i) {` `  trimmedLines[i] = x;`	`notEmpty.forEach(function(i) {` `  trimmedLines[i] = context;`
Express 3	ShiftOpChanger	`var key = req.accepts(keys);`	`const key = !! keys.length >> req && req.accepts(keys);`
Karma 4	FunctionMaker	`if (file && file.sourceMap )`	`if (file && (line || '').replace() )`
Mongoose 8	ReturnInserter	`this.constructor.update.apply(this.constructor, args)`	`return this.constructor.update.apply(this.constructor,args ,args )`
Bower 2	ArithmeticOpChg.	`if (that._options.save || that._options.saveDev )`	`if (that._jsonFile || this._manager * _direct )`

The first column is the unique identifier of the bug. Next to it the name of the operator is highlighted that created the patch which we found specific for JavaScript. The remaining columns list the original source line and the automatically generated patch. Code removals and additions are highlighted with red and green background respectively.

Table 2.13 shows some of the generated patches, which we found to be specific for JavaScript. Genetic operators in general does not take context into account: if an integer variable is compared against a number, an operator might replace the variable with another one, which holds a string value for instance. However, in case of repairing strongly-typed programming languages like Java or C++, the developer of the APR tool can take this information into account and design the operator to maintain the type of the variable as well. The types of the variables can also help to change conditional expressions, since most languages require the condition to be

a type of boolean or integer. In JavaScript, on the other hand, this intuition cannot guide the search process. For example comparing two strings or a string and an integer with the <= operator makes perfect sense here as can be seen in the first row of the table. In Table 2.13 in the rows of Eslint 72 and Karma 4 we can observe the use of the logical or (`||`) operator that in these cases serves as a kind of default value assignment. If we consider the case of Eslint 72, the assignment works as follows: first, the value of `penultimateToken.loc.end` is tested and if it is defined, the execution goes with it, otherwise the rest of the condition will be evaluated. Another aggravating circumstance for scripting languages is that it is much more difficult to examine the existence of an (overloaded) function. The example of Mongoose 8 demonstrates this phenomenon: the argument list of `apply` cannot be obtained and moreover it can be called with arbitrary number of parameters: if we do not specify some of the parameters they will be undefined, but the function call is syntactically correct. On the contrary if too many arguments are given, the leftover will not be taken into account in the function.

For script languages the search space is orders of magnitude larger than for compiled languages. This is mainly due to the fact that these languages are generally more permissive, giving operators more space to operate. Of course, if the operators are of such that do not leverage the restricting factors detailed above, the repair process is essentially the same for compiled and script languages (except for candidate validation offered by the compiler). However, the possibility to create more intelligent repair operators means a big advantage as the search space is definitely huge.

Another aspect of JavaScript program repair is its frequent interaction with the Document Object Model (DOM). In this work we could not focus on this kind of defects, since the BugsJS dataset does not contain bugs related to DOM manipulations. However most of the string manipulating operators are suitable for repairing such dynamic errors

Multi-line repairs

Batch operators have been designed with the aim to generate patches that span multiple lines. In addition the original ”genetic setup” is also implemented which allows the creation of multi-line fixes. Although the batch-operators are not able to generate multi-hunk fixes, we thought the tool could generate repairs more efficiently with these special operators (e.g. inserting a default branch in a switch-case and/or protecting a statement with a try-catch block could make sense). These approaches were constantly used in the search process. From the 31 generated patch **5** is a multi-line fix, thereby confirming that the approach is widely applicable. Since the human written developer fix is available for every bug in the dataset, we compared these with

the generated patches. We observed that there are cases when (1) the developer modified more lines than the GenProgJS tool and (2) the other way around: the generated patch includes multiple lines, while the developer fix is only a single line fix (usually it was a result of a batch operator and not the standard genetic approach). We found the case of (1) more interesting and depicted an example on Listing 4.1.

```
1   } else if(allowDangle === "always-multiline"){
2   >   lastTokenOnNewLine = node.loc.end.line !== penultimateToken.loc.end.line;
3   -    if(hasDanglingComma && !nodeIsMultiLine){
4   >    if(hasDanglingComma && !lastTokenOnNewLine){
5           context.report(lastItem, penultimateToken.loc.start,
    UNEXPECTED_MESSAGE);
6   +     hasDanglingComma = penultimateToken.value === ",";
7   -    } else if(!hasDanglingComma && nodeIsMultiLine){
8   >   } else if(!hasDanglingComma && lastTokenOnNewLine){
9           context.report(lastItem, penultimateToken.loc.end,
    MISSING_MESSAGE);
10
```

Listing 4.1: *(-) Original code of Eslint 7, (+) generated patch and (>) developer fix*

The key insight of Listing 4.1 is that the developer modified three lines in the patch, while the GenProgJS tool only one (the auto-generated patch is more concise than the developer fix). However the generated patch is clearly incorrect, the tests ran successfully (making it a plausible patch only). In our experiments those automatically generated patches which modified less lines than the developer were all incorrect ones. This of course, may be due to the work of chance or the unique characteristics of the BugsJS database.

Operator expedience

When to use which operator is of great interest, but a far-reaching conclusion can only be drawn with a sufficiently large database. The characteristics of the BugsJS database or the JavaScript language might imply the use of some operators, thus further examination needed in the field with the involvement of other databases and programming languages as well. We know that in our case each repair category is represented by a very limited number of bugs, but the correlation of some of the fix types and operators are easily visible and clearly explainable. Thus, we attempt to illustrate the reviewed correct patches in a figure and examine the relationship between operators and fix types.

The resulting chart can be found in Figure 2.20. Operators are listed on the vertical axis of the heatmap, while on the horizontal axis the repair types can be observed. Since the number of correctly repaired candidates varied widely, the usage

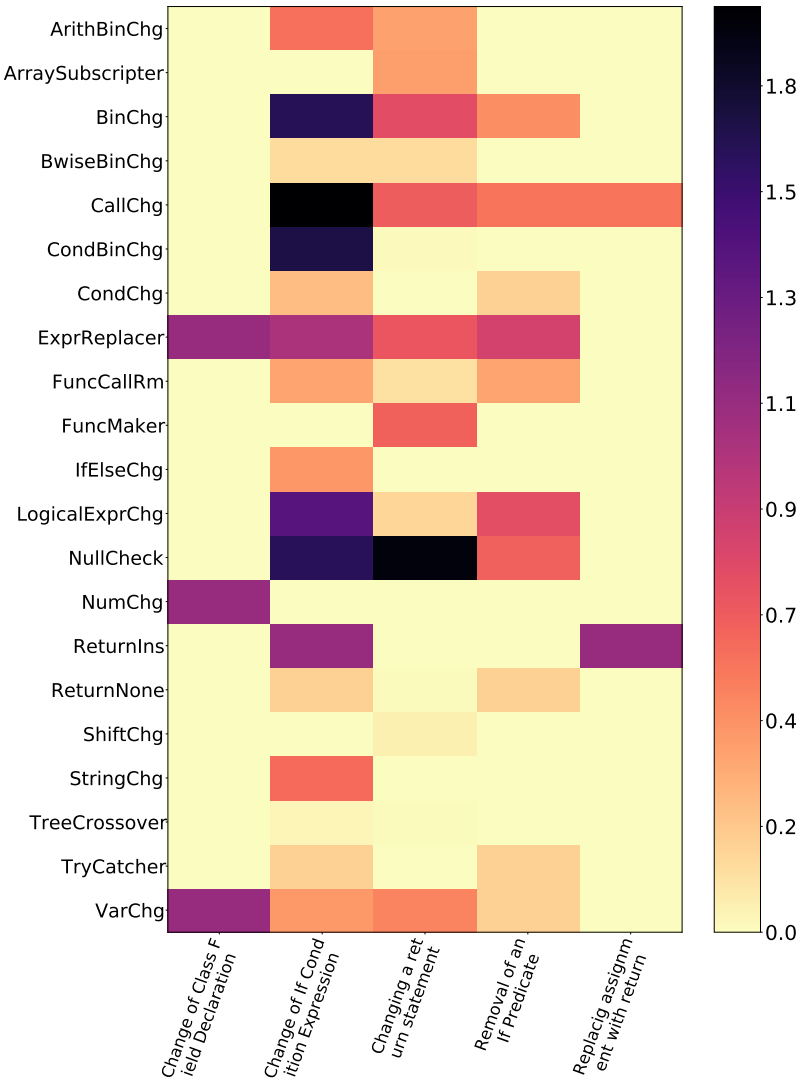


Figure 2.20: *Operator usage per fix type*

of operators was averaged per bug. That is for example, in case of a fix and two operators let us consider, that operator p_1 was used 10 times, operator p_2 was used 15 times, and a total of 10 candidates were generated, then p_1 would have the value of 1 on the figure, while p_2 would have the value of 1.5. For the same reason, on the heatmap not just integer values are indicated (which correspond to operator usages). The darker the color, the more times an operator was used for a given fix type.

What we can see is that for each fix type which operator made attempt to repair the bug (and actually during the process the bug was repaired). So for example in case of the fix type *Replacig assignment with return* only two operators are being used: *CallChanger* and *ReturnInserter* and it is clear that the latter one was used in most of the cases (its coloring is much darker). This connection seem to be quite obvious

and in fact it is, but highlighting the coherency of the operators and fix types further validates our work and may lead us to a better understanding of the whole process. The case of *Change of If Condition Expression* is however less self-evident: almost every operator made changes on the candidate patches. This clearly implies the fact that there were no single operator that was suited to repair this type of erroneous behaviour. The reason of this is however more nuanced. On the one hand, changing an if expression can involve a variety of changes; replacing an assignment with a return statement is quite straightforward, while changing an if expression is much more general. On the other hand, the operators which were used most of the time here make sense as well:

- *CallChanger*: is included in most types of repairs, we attribute this to the fact that functions are heavily used in JavaScript programs.
- *ConditionalBinaryOpChanger*, *BinaryOpChanger* and *LogicalExprChanger*: changing binary and/or logical operators in an if expression makes sense - this is basically the goal of these repair operators.
- *NullCheck*: it is obvious that checking the value of a variable also makes sense in and if expression. However thanks to the unique nature of JavaScript, this operator can be used in many scenarios - for example for default value assignment - probably that's why it is present in other fix types as well.

Defining the operator set that will generate correct fixes is impossible in practice, at least for now. As we saw on Figure 2.20 there are fix types for which it is rather easy to create a genetic operator, while for others it is nearly impossible to do so. Defining newer and newer operators might enhance the performance of some repair tools on a specific dataset (like BugsJS or Defect4J), however tailoring operators to databases is not forward-looking. The more operator is present in the repair process, the probability of each of them to operate will drop accordingly. One more complicating factor is that before creating a patch, in a real world scenario no one knows neither the type of the bug nor the type of the fix. Nevertheless, we can argue that in some cases (*e.g.* when the test case clearly fails under an if condition), it may be worthwhile to use certain operators in a targeted way. In these cases targeted operators more likely to synthesize a patch than others. Of course, the question arises about the use of completely generic operators such as *ExprReplacer* or *ExprStatementInserter*. These kind of operators are able to fix in theory any kind of bugs, but since the search space is large, they usually fail to find a plausible patch. Nevertheless, their use is definitely recommended, as only they are able to generalize to some degree in the traditional genetic approach. Finally we would like to emphasize that in this experiment we considered only those operator usages which resulted in a correct patch (and not plausible one).

Summary

With the following points the conducted experiments can be summarized:

- GenpPogJS successfully generated 31 plausible patches, which is comparable to the results of other languages.
- 5 plausible patches are identical and another 7 are semantically similar to the developer fix.
- Through the presentation of the generated patches, we described the unique nature of Automated Program Repair on JavaScript.
- The first patch of each bug is of prominent importance: both time-wise and operator-wise.
- Using genetic operator traces may highlight the weaknesses of some operators, the optimal usage of a given operator depends on the bug type.
- The generated plausible patches show variability in the types of bugs they address (14 types of bugs were covered).

2.8 Threats to validity

GenProgJS - G&V program repair

The experiments were conducted on a single bug dataset, which affects the generalization of the results. BugsJS contains actively maintained Node.js systems, and it is a manually curated dataset with dissection information, comparable to the Defects4J dataset, which is widely used in Java APR studies. We note that BugsJS contains server side Node.js projects, thus not intended to reason about client side repair problems. Generate and validate methods heavily depend on test suites, which are, in case of JavaScript, significantly better developed in server side projects [48]. Another threat is that this is the first work to study repairs on BugsJS, which is not directly targeted towards repair studies. At some points we needed to double check statistics and the test evaluation data provided.

The core process of GenProgJS is based on successful origins of GenProg and JGenProg, however it is adapted to the nature of JavaScript programs and uses an extended set of operators. We carefully examined several Java repair tools and checked whether their intended functionality is available in GenProgJS. In addition, we think that the repairs found by the algorithm validate the selection of operators. The implementation concentrated on the repair part of the process, thus missing fault localization. The analysis of the role of the operators is based on only one time runs

for each bug. We addressed this threat by emphasizing the role of the first plausible patch, since other patches many times are just extensions of the first one. However, a more targeted analysis is desired on this topic in the future. Finally, the parameters of the genetic algorithm are originated from the literature. JavaScript programs may require a different setup, which is to be found using a more exhaustive, systematic search.

Study on Fault Localization

A possible threat to validity of the FL research relates to the benchmark used. Only Defects4J has been utilized for evaluation, however it is the most commonly employed dataset in the literature for conducting similar studies. We believe that this choice does not impact the assessment of stability. Also, not all bugs from the benchmark were used in our experiments, due to hardware limitations. This does not limit the validity of the results, as if a model proves unstable even on this subset, it implies instability across the entire dataset. In this study we did not investigate the impact of low convergence levels, posing a threat to the validity of the findings. In scenarios when the primary objective is not generalization to new data, omitting a traditional train-test-validation split and convergence levels may align with the goals.

ChatGPT generated patches

ChatGPT was trained on a variety of text information, including source code. Despite the fact that this LLM was not fine-tuned to repair programs, it is quite effective in this task, albeit with some clear limitations. Since the language model used was also trained on source code, we cannot guarantee that the data used was not included in their training set. To address this issue, one would need to know exactly which repositories were included by OpenAI, and that information is not widely available. However, there are mitigating factors: (1) since ChatGPT was trained until a certain period of time, the data used by us is from a different version compared to the data OpenAI might have used; (2) although our dataset satisfies our evaluation criteria, it constitutes only a tiny fraction of the training data used by OpenAI.

Although choosing the right prompt is of great importance, in this paper, we did not provide guidance on how to approach the choice of prompts. During our experiments, we observed that a certain prompt triggers a specific repair mechanism, and if these templates could be mapped to bug types, it would guide developers on how to choose the right prompt. ChatGPT (specifically the model in the correct version) is not openly available, unlike some other LLMs, so reproducibility cannot be ensured. Since the model is exposed via a UI and an API, OpenAI can change the model at any time, even without the user knowing it. To address this limitation, we included input/output prompt samples in the online appendix of this chapter.

2.9 Concluding remarks

In this chapter the usage of Machine Learning is discovered in the domain of Automated Program Repair. First, the application of Deep Learning is discussed in the start of the repair process: classical Fault Localization methods still seem to be more reliable than ones supported by AI. Next, a dataset creation process is showcased which helps in the training and evaluation of repair techniques. Patches are generated using two quite distinct approaches: genetic algorithm and a transformer model. Both of these were able to generate correct fixes, although overfitting is still a major issue in the field.

In the domain of FL, the work only concentrated on the issue of stability; however, this might be a general problem in SE research using DL. DL reproducibility can be largely supported by sharing a reproduction package, however without examining the stability of the proposed model, its applicability is limited. As we have seen, standard statistical measures serve as a good indicator of whether a model is stable or not, and the adapted churn metric can further support such arguments. By applying the proposed method, we believe better practices can be built for publishing DL applications and testing their stability. Another question that arises is what can we conclude about the reasons for the instability and how it can be mitigated. The major problems which we managed to moderate were the imbalanced data, suboptimal network architecture and parameterization, and in general, suitability of DL for SBFL. The implications of this research are twofold. First, SBFL research employing Machine Learning techniques should pay much higher attention on model stability because it decisively impacts the soundness and significance of the results. Second, other SE fields employing DL could also benefit from our stability measurement approach.

State-of-the-art APR approaches are typically evaluated on their own datasets which are often not publicly available, which hampers the comparative evaluation of novel methods. In part of this chapter, the FixJS dataset is described, which includes ˜300k samples containing separately the buggy and fixed codes. During the dataset creation process ˜2 million commits were examined and ˜200k JavaScript files mined. From this massive amount of data three datasets of different sizes were created: small, medium and large. In each of these datasets three source code representations with different abstraction levels are present. FixJS is of similar size like in [168], but it only contains commit info for a limited time interval. Although the two datasets operate on different programming languages, tools evaluated on them are might be comparable.

The presented program repair tool, GenprogJS, is based on a genetic algorithm and targets buggy programs written in JavaScript. It is designed relying on the well-known tools for the C and the Java languages, but taking into account the properties

of the language it is adapted for. The motivation for this work is that to our best knowledge there is no such generic test-based repair tool available for JavaScript. According to the first experiments GenProgJS found plausible repairs for 31 bugs in 6 Node.js projects, which is a comparable result to the related work done on other languages. We provided a detailed analysis of the applicability of genetic operators, as well as presented examples of plausible patches that found by the algorithm. We provide this work and the belonging GitHub repository to facilitate APR research and hope that this could serve as a baseline for future work on JavaScript programs.

The capabilities of ChatGPT were also investigated in the field of APR, specifically how it performs when tasked with fixing buggy code. 200 buggy codes were sampled from seminal APR datasets, consisting of 100 Java and 100 JavaScript samples. 5 input prompts were designed for ChatGPT. The results demonstrate that these prompts have a significant effect on the repair performance, as different prompts trigger different repair mechanisms of the LLM. The overlap of the fixed bugs is negligible. Through manual evaluation of the outputs, we observed that better repair candidates are generated for Java compared to JavaScript. The best prompt for Java generated correct answers in 19% of cases, while for JavaScript, the same prompt yielded a performance of only 4%. In total, 44 distinct bugs were repaired in Java and 24 in JavaScript out of the overall 200 samples and 1000 repair trials. We found that some bugs are more likely to be correctly patched with a well-chosen prompt rather than a poorly chosen one. Therefore, the most important question before starting the repair process is to select the appropriate prompt. ChatGPT may be a useful tool for improving software reliability in practice, but one should not blindly trust the code it generates. ChatGPT appears equally confident when generating correct code as it does when generating incorrect code.

The author of this PhD thesis is responsible for the following contributions presented in this chapter:

II/1. Adaptation of churn to FL.

II/2. Design of experiments in SBFL, neural network optimization approaches.

II/3. Development of the FixJS mining process, dataset curation and publication.

II/4. Implementation of the genetic algorithm in GenProgJS.

II/5. Design and coordination of experiments in genetic program repair.

II/6. Prompting and evaluating ChatGPT in the sampled datasets.

3 Automated Assessment of Automatically Generated Patches

3.1 Overview

Automated Program Repair (APR) strives to minimize the expense associated with manual bug fixing by developing methods where patches are generated automatically and then validated against an oracle, such as a test suite. However, due to the potential imperfections in the oracle, patches validated by it may still be incorrect. Several approaches have been proposed that use a variety of information from the project under repair, such as diverse manually designed heuristics or learned embedding vectors. The predominant focus of APR research revolves around Generate-and-Validate (G&V) approaches, wherein patch candidates are generated (e.g., via genetic algorithm, heuristics, or learned code transformations) and subsequently verified against an *oracle*. If the oracle is the test suite (which is usually), the approach is referred to as test-suite-based program repair. Despite facing criticism on multiple occasions, these methods still shape the trajectory of APR research [83]. A notable obstacle encountered in test-suite-based repair is the potential to create a patch that enables the entire test suite to pass, yet remains incorrect. This phenomenon is commonly referred to as the overfitting patch problem [172] and the goal of Patch Correctness Check (PCC) is to determine the actual correctness of a patch, without additional manual effort.

The generation of overfitting patches leads to the generation of program repair patches with limited utility, thereby substantially affecting the practical applicability of program repair. It also makes developers less confident in APR tools, thus reducing their widespread use. The use of data-augmentation techniques [185, 195], and repair operator curation [176] can lead to more correct patches, but at the time of writing this dissertation, the classification of generated patches is the most popular research direction. Among these recent studies have introduced static methods for detecting overfitting patches, mainly because of their ease and speed of use. Xin *et al.* [186] defined 2 static features in ssFix, while 3 static features are leveraged in S3 [94] and 202 in ODS [194]. Another approach is the use of source code em-

beddings directly [105]. Most of these works operate on distinct datasets, and approaches are in competition with each other and not complementary. My aim is to handle all of these crafted knowledge as *features* for a machine learning model and select which of these are the most useful in the PCC domain. In addition, similarity measured on the embedding vector is also introduced as a standalone technique and as a feature as well.

In the following two experiments are executed and evaluated. First, it is investigated whether similarity is an appropriate approach to tackle with the PCC problem. To do so, similarity between generated plausible patches and the original code is measured. The intuition behind similarity-based approach is that more similar patches deem to be more simple. Using source code embeddings allow us to place these textual documents into some high-dimensional space, where usual similarity measures can be applied. For example considering an APR patch that lies closer to the developer patch than the other patches, it might be better in some way. Of course in higher dimensions, the similarity metric should be well defined, and the exact meaning of each dimension can not be interpreted in most cases. In the following similarity is measured between the generated patches and the original program and also included the developer fix in the comparison process. The source code similarities are measured using document/sentence embeddings, specifically with two state of the art techniques borrowed from the natural language processing domain: Doc2vec [140] and Bert [40].

Next, 903 patches generated by APR tools in Defects4J [81] and previously labeled by researchers [172] are used for classification purposes. For these patches features are mapped achieving state-of-the-art results in previous works: hand-crafted features [94, 176, 186], static code features [194], embedding vectors [105] and the introduced similarity metrics [31] - thus forming a feature vector of 490 dimension. On this set, a feature selection process is performed, which resulted in a 43-dimensional feature vector. The selected features served as the basis for the model selection, in which ML models are trained and selected in which Multi-Layer Perceptron (MLP) yielded the overall best performance.

3.2 Related Work

Evaluating existing Automated Program Repair approaches is crucial, but assessing APR tools solely on plausible patches is inaccurate due to the overfitting issue inherent in test suite-based automatic patch generation. Identifying the correct patches among plausible ones requires additional developer effort. Recently several approaches have been proposed to tackle the problem of Patch Correctness Check (PCC) [193].

Generation of correct patches

To generate repair patches as simple as possible, has already mentioned in many works [121, 168, 177]. This makes the repaired programs more understandable to humans. Such codes that are generated by APR tools without any effort to make them readable are called ”alien code” [127]. Although, their subsequent maintenance may be costly, according to a recent study [171] 25.4% (45/177) of the correct patches generated by APR techniques are syntactically different from developer-provided ones. Angelix is a semantics-based test-driven automated program repair tool for C programs, which is capable of producing multi-line fixes, that are less prone to deleting functionality [122]. The Hercules tool repaired 15 multi-hunk bugs in the Defects4J dataset, which is an important contribution towards generalizing APR. An important aspect of the future of program repair is deciding the correctness of candidate patches [11]. In [52] authors highlighted this issue as an open question. Previous works [159] has pointed out that too many test cases are not beneficial in the field of automatic program repair, as this is when the problem of overfitting typically occurs. It is also a known phenomenon that there are errors that remain hidden under ”laboratory conditions” [71]. Nevertheless, the significance of test cases is particularly important, as in some cases even the creation of patches is based on tests [159, 192]. Other approaches also exists, which generate patches by learning human-written program codes [87, 95]. While such approaches have shown promising results, they have recently been the subject of several criticisms [126]. Evaluating APR tools based on plausible patches are not accurate, due to the fact of the overfitting issue in test suite-based automatic patch generation. Finding the correct patches among the plausible patches requires additional developer workforce. Liu *et al.* [108] proposes eight evaluation metrics for fairly assessing the performance of APR tools beside providing a critical review on the existing evaluation of patch generation systems.

Feature-based PCC

In a recent study [172] benchmarks the state of art patch correctness techniques based on the largest patch benchmark so far and gathers the advantages and disadvantages of existing approaches beside pointing out a potential direction by integrating static features with existing methods. Another work where embedding methods were used for ranking candidates is [165], where beside Doc2Vec and Bert, code2vec and CC2Vec were also applied. In this work they investigate the discriminative power of features. They claim that Logistic Regression with BERT embedding scored 0.72% F-Measure and 0.8% AUC on labeled deduplicated dataset of 1,000 patches. Liu *et al.* [108] proposes eight evaluation metrics for fairly assessing the performance of APR tools in addition to providing a critical review on the existing evaluation of patch generation systems. Opad [192] (Overfitted Patch Detection) is another tool which aims to filter out incorrect patches. Opad uses fuzz testing to generate new test cases

and employs two test oracles to enhance the validity checking of automatically generated patches. Anti-pattern based correction check is also a viable approach [164]. Syntactic or semantic metrics such as cosine similarity and output coverage [94] can also be applied to measure similarity, like in the tool named Qlose [37]. A recent study [76] presents a new lightweight specification method that enhances failing tests with preservation conditions, ensuring that patched and prepatched versions produce identical outputs under specific conditions.

Dynamic PCC

Numerous studies [121, 168, 177] emphasize the importance of simplifying the generated repair patches. A recent study [171] found that 25.4% (45/177) of correct patches generated by APR techniques differ syntactically from those provided by a developer. Other methods, such as learning from human-written code [87, 95], have shown promise but have faced recent criticism [126]. In other works [29] candidate patches were ranked according to their similarity to the original program and assessed as a recommendation system. Others have also used embedding techniques, but not only focusing on the changed code, but also taking into consideration the unchanged correlated part [105] by measuring the similarity between the patched method name and the semantic meaning of body of the method [142] or based their approach onthe fact that similar failing test cases should require similar patches [166]. The reliability of automated annotations for patch correctness has also been proposed [93]. Authors compared them with a gold standard of correctness labels for 189 patches, finding that although independent test suites may not suffice as effective APR oracles, they can augment author annotations. Meanwhile, Xiong *et al.* [187] proposed leveraging behavior similarity in test case executions to determine correct patches. By improving test suites with new inputs and using behavior similarity, they prevented 56.3% of incorrect patches from being generated. Syntactic or semantic metrics like Cosine similarity and Output coverage [94] can also be applied to measure similarity, like in the tool named Qlose [37]. These metrics have several limitations, like maximal lines of code to handle or that they need manual adjustment. On the other hand, the use of document embeddings offers a flexible alternative. Opad [192] (Overfitted Patch Detection) is another tool, which aims to filter out incorrect patches. Opad uses fuzz testing to generate new test cases and employs two test oracles to enhance validity checking of automatically generated patches. Anti-pattern based correction check is also a viable approach [164].

This work

In a recent study [93] authors assessed reliability of automated annotations on patch correctness assessment. They constructed a gold set of correctness labels for 189 patches through a user study and then compared these labels with automated generated annotations to assess reliability. They found that independent test suite

alone might not serve as an effective APR oracle, it can be used to augment author annotation. In the paper of Xiong et al. [187] the core idea is to exploit the behavior similarity of test case executions. The passing tests on original and patched programs are likely to behave similarly while the failing tests on original and patched programs are likely to behave differently. Based on these observations, they generate new test inputs to enhance the test suites and use their behavior similarity to determine patch correctness. With this approach they successfully prevented 56.3% of the incorrect patches to be generated. In this work, I mainly focused on the optimization of the features used in ML models and partly with enhancing the performance with deep learning. The data used is partly from the study of Wang *et al.* [172] and the engineered features from ODS [194]. The work is fundamentally different from previous onw: (1) I defined a new similarity-based feature together with others existing ones, (2) all available features are treated as complements to each other, (3) the goal was to achieve a cross-research feature set which is the most optimal to PCC. In a recent study, Tian *et al.* [167] already proposed deep-combination of features, but their approach (1) does not apply feature selection, (2) no diverse ML model training is carried out, and (3) the used static feature set is not as thorough as the ones we have presented. Here, the training dataset along with the fixed seeds and the source code of the experiments is also published (which is only partially true for previous studies).

3.3 Method

Traditionally, a patch is deemed correct if it successfully passes all the test cases, unfortunately practical test suites often fail to guarantee the accuracy of generated patches. Consequently, patches that pass all tests (referred to as plausible patches) may incorrectly address the bug, fail to fully fix the issue, or disrupt intended functionalities, thus becoming overfitting patches [112]. In the following sections two methods are described: first an embedding based approach, where the similarity between the original code and the generated patches is the only decisive factor in the classification task (correct / incorrect patch). Next, other code features are also considered (alongside with embeddings) and multiple ML classifiers are trained to support the decision.

3.3.1 Using similarity in PCC

In this section the used approach is described to determine the usefulness of similarity based patch validation. A high-level overview of the proposed process can be found in Figure 3.21. First an APR tool creates plausible patches, usually more than one. In this case the tool always ran for 30 generations resulting in a high number of

plausible patches for each bug. From the original program and from the generated potentially fixed programs, the faulty line is extracted and a small environment of it - this snippet of code will serve as a basis for calculating similarity. After a Doc2Vec model is trained, for every code snippet an N dimensional vector is created on which one can measure similarity. The generated plausible patches then lined up alongside with the developer fix, based on the similarities calculated previously. Based on the list we can analyze which version of the fixed program is the most similar to the original one - the one created by an APR tool or a developer fix.

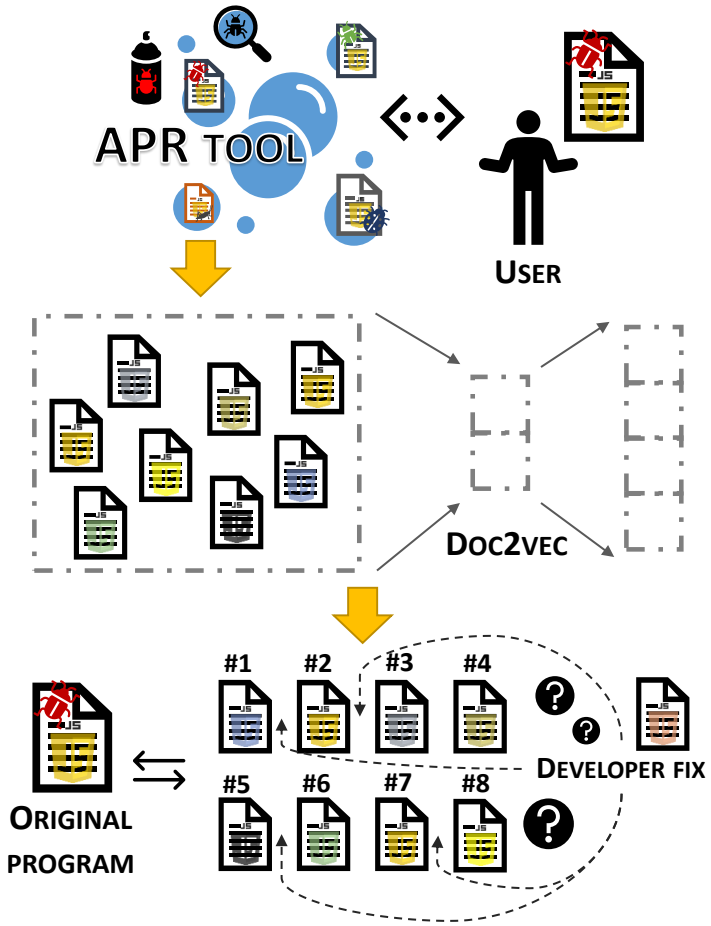


Figure 3.21: *Illustration of the implemented similarity-based process.*

Ranking plausible patches alongside with the developer patch may point out that a human written patch differs greatly from the original source code. This does not necessarily mean that the developer made a mistake: it might be that he adapts a new approach that was never used before in the code base, or simply made a refactoring of considerable size.

Learning Document Embeddings

For every bug a small environment of the faulty line was selected, and this was embedded using Doc2Vec (detailed description of it can be found in Chapter). This code environment includes the faulty line itself and three lines in front of and behind it. For the training data this small code fragment is first tokenized with a simple regular expression, which separates words and punctuations, except for words with the dot `'.'` (member) operator. For a simple code example like: `function foo () { return this.bar; }` the tokenized version would be: `'function', 'foo', '(', ')', '{', 'return', 'this.bar', ';', '}'` . In Doc2Vec a window size of 5 was used, which tells the model the maximum distance between the current and predicted word. Every word with a frequency of less than 2 was ignored. As for the training, the model was trained for 50 epochs. Every other parameter was left as default. On the obtained embeddings (vectors containing real numbers) similarity is measured with the *COS3MUL* metric, proposed in [102]. According to the authors positive words still contribute positively towards the similarity, negative words negatively, but with less susceptibility to one large distance dominating the calculation.

Evaluation of the similarity list

The main question is, whether it is true — from the perspective of Doc2Vec — that the developer fix lies close to the original program. Current state-of-the-art APR applications still fail to repair real complex issues, thus the demand for simple patches may be desirable. To measure the quality of the ranking, we used the Normalized Discounted Cumulative Gain (nDCG) metric, which is computed as:

$$nDCG_p = \frac{DCG_p}{IDCG_p} \tag{3.7}$$

Where *DCG* stands for Discounted Cumulative Gain, *IDCG* stands for Ideal *DCG* and *p* is a particular rank position. *DCG* measures the usefulness, or gain, of a document based on its position in the result list. *IDCG* basically is the maximum possible *DCG* value that can be achieved on a ranked list - this is done by sorting all relevant documents in the corpus by their relative relevance. Since the similarity lists vary in length (the number of plausible patches is different for each bug), consistently comparing their performance with *DCG* is not possible. To be able to compare these lists, cumulative gain at each position for a chosen value of *p* should be normalized, thus resulting in the nDCG metric defined above in Equation 3.12. The definition of *DCG* and *IDCG* is presented in Equation 3.8 and Equation 3.9 respectively.

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)} \quad (3.8) \qquad IDCG_p = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{log_2(i+1)} \quad (3.9)$$

The rel_i is the graded relevance of the result at position i. Since the similarity values give us the ordering, each item in the list should have another value which validates its placement. We manually checked each and every generated bug and categorized them based on their relevance. The following relevance scores were introduced:

- 3: the developer fix always has the highest relevance, in ranking the most favorable is when this patch comes first
- 2: the patch syntactically matches the developer fix - we use the term syntactic match when the codes are the same character by character, apart from white spaces
- 1: it is semantically identical to the developer fix - that is, the two source codes have the same semantical meaning, but there may be character differences
- 0: we were uncertain about the patch
- -1: the patch is clearly incorrect (e.g. syntactic errors)

In addition to these, intermediate categories are also conceivable: e.g. -0.5 would mean that a patch is probably incorrect, but because of the lack of domain knowledge about the examined system it is undecideable. Two experienced software developers separately annotated the generated patches, they did not have the chance to influence each other. In cases where individual scores differed a third expert decided on the correctness of the patch. These annotated relevance scores are available in an online appendix [7].

3.3.2 Feature-based PCC

In this section the usefulness of features proposed in previous works are investigated, to tackle the problem of identifying correct patches among incorrect and plausible APR-generated patches. First the necessary background is provided on the used engineered and learned features.

[7] https://github.com/RGAI-USZ/JS-patch-exploration-APR2021

Features used in Classification

Generally, within Machine Learning, a feature represents a distinct and measurable attribute or characteristic of a phenomenon, while a feature vector refers to real numbers in an n-dimensional space, consisting of features [157]. Features are widely used in Software Engineering for diverse tasks including just-in-time quality assurance [82], fault localization [89], vulnerability prediction [36] and others. In this work, we focus on optimal feature selection in the domain of Patch Correctness Check. The overview of these features van be found in Table 3.14.

Hand-crafted Features

Manually crafting features so that a classifier can prioritize correct patches over overfitting ones is not uncommon in the literature. Features of ssFix [186], S3 [94] and CapGen [176] are adapted, as their implementation is available and because these values are available in the seminal work of Wang *et al.* [172]. The features introduced in S3 quantify both syntactic and semantic disparities between a candidate solution and the original buggy code. Subsequently, these features are used to prioritize and discern correct patches. In ssFix, authors employ token-based syntax representation of code to pinpoint syntax-related code fragments, aiming to generate accurate patches. CapGen has proposed three context-aware models - the genealogy model, variable model, and dependency model, respectively - to prioritize correct patches over overfitting ones.

Engineered Features

ODS [194] is also used to extract metrics from the source code. It performs static analysis of the differences in AST between the buggy and patched programs; these differences are encoded as feature vectors. The authors grouped the ODS features into three categories: code description features, repair pattern features, and contextual syntactic features. We extracted these features using Coming [8], an open source commit analysis tool. Due to space limitations, not all features are described here (they are available in the original study), just an overview in Table 3.14.

Table 3.14: *The used PCC features to classify overfitting patches.*

Category	Feature	Origin	Description	# dim
Hand-crafted	Token- Strct/Conpt	ssFix [186]	Structural & conceptual token similarity obtained from the buggy code and the generated patch.	2
	AST/Variable- Dist	S3 [94]	Number of the AST changes and distances between the vectors representing AST patch nodes.	4
	Var/Syn/Sem- Simi	CapGen [176]	Similarity between variables / syntactic structures / contextual nodes affected by the change.	3
Engineered	Code descriptor	ODS [194]	Describe the characteristics of code elements (operators, variables, statements, AST operations)	155
	Repair pattern	ODS [194]	Repair patterns based on the work of Sobreira *et al.* [160] as binary features.	38
	Contextual Syntactic	ODS [194]	Describe the scope, parent and children's similarities of modified statements.	24
Dynamic	Embedding	Cache [105]	Dimensions of the embedded patch.	256
	Similarity	[31]	Similarity metrics measured between the embeddings of the patch and the original program.	8
				490

[8] https://github.com/SpoonLabs/coming

Code Embeddings and Patch Similarity

Mapping source code into vector space can be beneficial in many ways, it can grasp aspects of the program that other metrics cannot. In the domain of patch correctness assessment, these techniques assess patch correctness by embedding token sequences extracted from the changed code of a generated patch [105]. Here again, Doc2Vec is used to gerenate embeddings, detailed in Chapter . The embeddings can be used in various ways; two scenarios are considered in this study.

1. Use the generated embeddings directly, as if a dimension of the embedded vector is a feature. Thus, each dimension is treated as a metric measured on the patch.

2. Measure the vector distance between the embedded vector of the patch and the original program. The intuition here is the same as described in Section 3.3.1

Experiment Setup

Figure 3.22 shows a comprehensive overview of this classification task. The goal is straightforward: find the set of features and ML models that most effectively detect overfitting patches. In Figure 3.22 (a) one can see that features are concatenated as is and then selected some of them, while (b) part depicts a deep representation model where features are fed into a neural architecture, allowing the net to learn the weights and biases of each feature. The obtained features from previous studies, described in Section 3.3.2, form a feature vector of l=*480* dimensions (composing of both static and dynamic features). Using feature selection techniques, l' features are selected from these (l' *i* l) - the ones that best explain the input data. ML models are trained and evaluated on this subset to determine which yields the most optimal results. In the experiments a 32-bit Intel(R) Core(TM) i7-10510U CPU of 1.80GHz was used to train and evaluate each model and feature configuration. All of the code runs in Python 3.11.6, using the scikit-learn library version 1.4.0 [138]. The source code and detailed experimental data can be found in the online appendix [5].

Feature Selection

Feature selection is the process of selecting a subset of relevant features to be used in model training [154]. There are many available feature selection algorithms, from which the scikit-learn implementation of RFECV was used [150] to achieve the goal depicted in Figure 3.22 (a). It recursively eliminates features with cross-validation to select the most important features. The number of features selected is tuned automatically by fitting an RFE selector to the different cross-validation splits. A *RandomForestClassifier* was used as an estimator to provide information about feature importance mainly because it is a preferred model in previous PCC studies [172, 194]

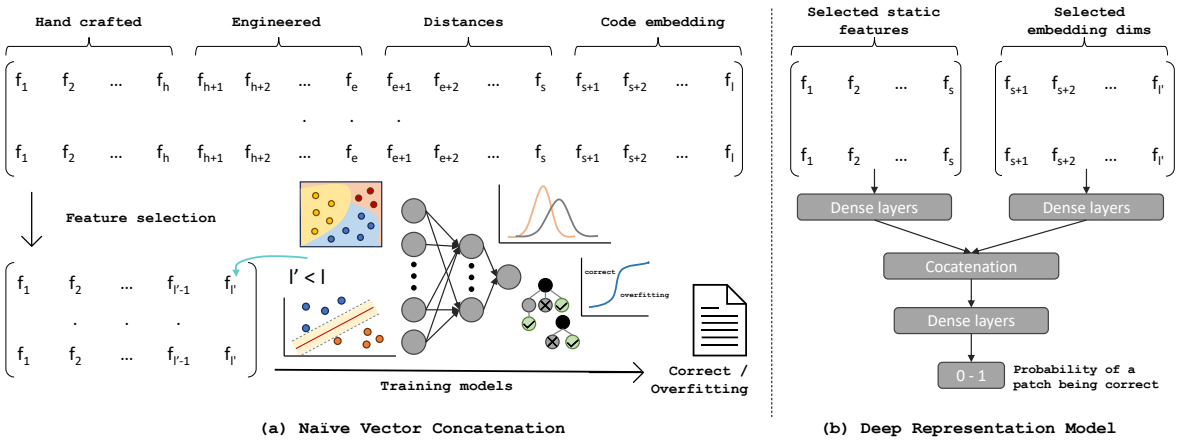


Figure 3.22: *A high level overview of the used features and their optimization for PCC. On part (a) all features are concatenated then the most descriptive ones are selected to teach several ML models. On (b) static features (Hand-crafted, Engineered and Distances) and embeddings are first fed into dense layers and the neural network concatenates them, allowing it to learn a dynamic representation.*

and it has easily accessible coefficients required by the feature selection algorithm. Experiments were carried out in a *Stratified K-Fold* setting using 10 splits and the minimum number of features was required to be 10.

ML models

The utilization of scikit-learn is motivated by its accessibility, robust performance, and inclusion of well-established, reliable models, facilitating the execution of our experiments with ease and efficacy. The following 10 models were used in part of our experiments (Figure 3.22 (a)): DecisionTreeClassifier, GaussianNB, KNeighborsClassifier, LinearDiscriminantAnalysis, LogisticRegression, MLPClassifier, RandomForestClassifier, LogisticRegression, SGDClassifier and SVC. The description each of these models can be found the official documentation of the scikit-learn library [155]. The concatenated features form the input for these models.

To further enhance the performance, in addition to built-in ML models, a neural network was built using Pytorch 2.1.2. It is able to combine features with learned embeddings as suggested by Tian *et al.* [167]. The approach can be observed on Figure 3.22 (b). Note that this model operates on the already selected features, but treats embeddings dimensions and numeric features separately and concatenates them dynamically. The gist of this approach is that the neural architecture can learn the weighting of each feature and is able to inference more complex relations compared to naive concatenation.

Evaluating the Classification Task

Previous studies have underscored the importance of PCC techniques in avoiding the dismissal of correct patches (as they are quite expensive to generate in the first place) [197]. Consequently, a PCC technique is deemed effective if it produces minimal false positives while maintaining a high recall rate. To quantify the classification results, the following metrics are introduced:

- **T**rue **P**ositive: An overfitting patch is correctly identified.
- **F**alse **P**ositive: A correct patch identified as overfitting.
- **F**alse **N**egative: An overfitting patch identified as correct.
- **T**rue **N**egative: A correct patch identified as correct.

Using the above items, precision, recall, and F-meaure can be computed. Precision is the proportion between correctly classified overfitting patches among all the classified instances, while recall is the proportion between correctly classified overfitting patches and all relevant items. They are computed as:

$$precision = \frac{TP}{TP + FP} \quad (3.10) \qquad recall = \frac{TP}{TP + FN} \quad (3.11)$$

The F-measure can be defined by the two metrics above:

$$F_\beta = \frac{(\beta^2 + 1) * precision * recall}{\beta^2 * precision + recall} \quad (3.12)$$

β signifies the importance of precision or recall. If we want precision and recall to weigh in with exact the same importance, we simply assign the value 1 to β.

3.4 Datasets

First, the results of similarity-based approach is presented. To show the effectiveness of this approach, for a given bug several generated patches need to be present. To the best of my knowledge no such dataset exists, thus the previously presented GenProgJS tool was used to generate patches on the BugsJS dataset. This process is detailed below. To evaluate feature-based PCC we used a widely adapted and curated Java dataset, for which the previously showcased features are their calculations are already implemented and accessible.

3.4.1 Sample Plausible Patches to Measure Similarities

The BugsJS dataset [64] containes 453 reproducible JavaScript bugs from 10 open-source Github projects. The dataset contains multi-line bugs as well, which are beyond the scope of the current research. There are 130 single-line bugs, but not every one of them are ”repairable”, because of the lack of failed test cases. Thus, the number of single-line bugs, for which there is at least 1 failed test is 126 (and 94 only comes from the Eslint project). BugsJS features a rich interface for accessing the faulty and fixed versions of the programs and executing the corresponding test cases. These features proved to be rather useful for the comparison of automatically generated patches with the ones which were created by developers. Experiments were limited strictly to the Eslint project because it is the largest project in the BugsJS dataset, it contains the most single-line errors. The automatic repair tool which was used was able to repair 10 bugs from 94 in the Eslint project. Since the tool was configured to run for 30 generations in every case (so it does not stop at first when a fix is found), there was a high number of repair candidates in most cases of the runs as can be seen in Table 3.15. In the first column one can find the id of each bug and next to it how many plausible patches were generated to it. The two remaining columns show the original source code and a fix for it created by a developer.

Table 3.15: *Plausible patches and their corresponding developer fix in the Eslint project*

Bug Id (Eslint)	No.	Original line	Developer fix
1	4	`if (name === "Math" || name === "JSON")`	`if (name === "Math" || name === "JSON" || name === "Reflect" )`
41	3	`end.column === line.length )`	`( end. line === lineNumber && end. column === line.length ));`
47	3	`column: 1`	`column: 0`
72	7	`loc: lastItem .loc.end,`	`loc: penultimateToken .loc.end,`
94	14	`op.type === "Punctuator" &&`	`( op.type === "Punctuator" || op.type === "Keyword") &&`
100	12	`penultimateType === "ObjectExpression"`	`( penultimateType === "ObjectExpression"` `|| penultimateType === "ObjectPattern")`
217	4	`if (!options || typeof option === "string")`	`if (!options || typeof options === "string")`
221	221	`return parent.static ;`	`return false ;`
321	5	`...loc.end.line !== node.loc. end .line &&...`	`...loc.end.line !== node.loc. start .line &&...`
323	192	`else if (definition.type === "Parameter"` `&& node.type === "FunctionDeclaration")`	`else if (definition.type === "Parameter")`
Total	465		

We can see that there are bugs of different difficulty in Table 3.15: from quite simple where a number had to be replaced (Eslint 47), to quite complex where a conditional expression needed to be supplemented (Eslint 100). The number of generated candidates also varies greatly, this is due to the difficulty of the fixes and the random factor in the GenProg algorithm. In total 465 plausible patches were gener-

ated. These patches were checked and found that only three of these are syntactically identical to the developer fix (Eslint 47, Eslint 323 and Eslint 72), although many of them are semantically identical.

It is apparent that for Eslint 221 and Eslint 323 the number of plausible patches is orders of magnitude more than for any other. To explain it let us examine the nature of these bugs. The case of Eslint 221 is quite easy to understand: the return value should be `false`, making it rather simple to generate. We examined the generated patches and found that essentially anything would satisfy this criteria: in JavaScript `0`, `-0`, `null`, `false`, `NaN`, `undefined`, or the empty string (`""`) create an object with an initial value of `false`. On the other hand in case of Eslint 323 the high number of plausible patches is most probably because of the weak test suite. As we can see from Table 3.15 the fix is not quite obvious, but after carefully inspecting the generated fixes we came to the conclusion that every modification on which the `if` condition evaluated to `true` successfully passed testing.

3.4.2 Dataset on feature-based PCC

Wang *et al.* [172] published a curated dataset comprising 902 labeled patches from Defects4J bugs, generated by 19 repair tools. This dataset is being used, which encompasses 654 patches labeled as overfitting and 248 patches labeled as correct by the respective authors. The dataset is still actively maintained, easily available, and popular to this day. In their online appendix [9], authors have also published the measured hand-crafted features on this dataset which we used directly from there. ODS features are calculated using their tool, while dynamic features are calculated using our implementation available in the attached repository [5].

3.5 Results

3.5.1 Similarity-based Evaluation

The calculated metric values of nDCG described in Equation 3.12 can be found in Figure 3.23. We can see that in case of Eslint 217 and Eslint 41 the values are 1.0, this is clearly because the developer fix was ranked to the first place in these cases and irrelevant documents were placed on the end of the similarity list. Based on this metric it is clear that in most cases similarity lists hold their place in ranking patches. The nDCG metric value reaches its lowest point at the Eslint 94 bug. If we take a look at the subplot (e) at Figure 3.24 we can see that indeed Doc2vec failed to rank the developer fix at the top of the list. Moreover, most of the patches at the top of the list are incorrect ones, meaning that they hold low relevance. The case of Eslint 100

[9] https://github.com/claudeyj/patch_correctness/

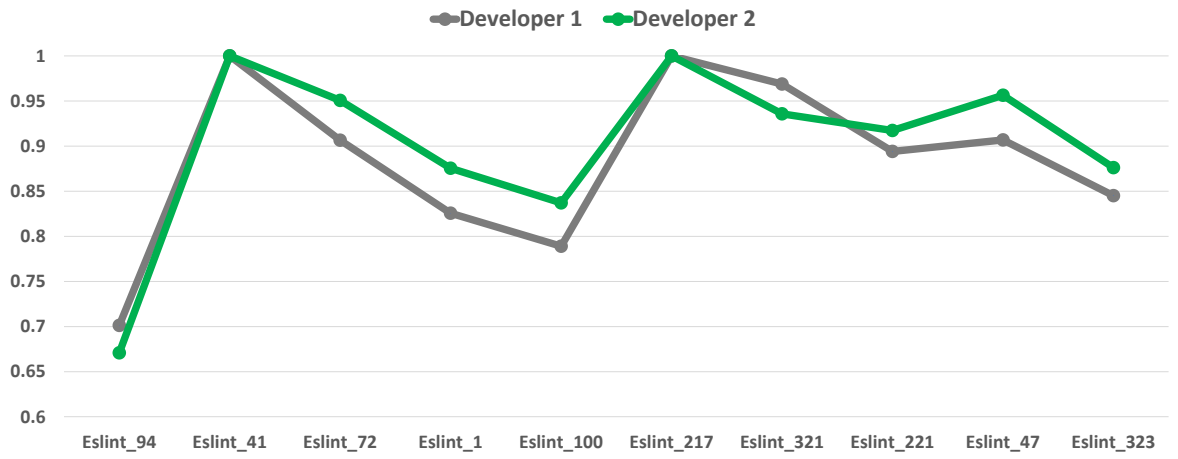


Figure 3.23: *The values of nDCG based on the two developer evaluation. The possible values of the metric ranges from 0.0 to 1.0, a higher metric value means better ranking.*

is different from the previous one. Although it is true that the nDCG value of Eslint 94 is 0.67, while for Eslint 100 it is 0.84, compared to others it still seems to be quite low. If we take a look at the rankings at Figure 3.24 we can see that in this case the developer fix is placed on the second place of the ranked list. So the question arises, what causes the low metric value? The answer is quite obvious: the patch which is placed ahead is an incorrect one, decreasing the metric value drastically. The case of Eslint 221 is also interesting: although the developer patch is placed closer to the end of the list than anywhere else, the nDCG metric value is not that low. This is due to the fact that in case of this bug the majority of generated plausible patches are semantically the same as the developer fix, resulting in overall higher relevance scores.

The produced similarity lists can be observed on Figure 3.24. It is cleary visible that in most cases the developer fix has been placed on a prime location in the similarity list. The developer fix is at the top of the list in 3 cases and takes the second place in 4 cases. Note that the ranking of the lists are quite instable for two reasons: (1) the numerical difference is not outstanding between each similarity value, and (2) Doc2Vec fails to give back identical similarity value even though the same documents are compared. Because of these previously mentioned limitations different Doc2Vec model trainings can even result in completely distinct lists of similarities.

In Listing 5.1 one can examine the original line with red background and the line that was generated by an APR tool with green background. In this case the developer fix adds another logical testing in the if condition, allowing the `name` variable to have the value `"Reflect"` as well. The modifications which the APR tool made bears no resemblance to this. At first sight it does show greater similarity with the original line than the developer fix, however the generated code is clearly not the best. First

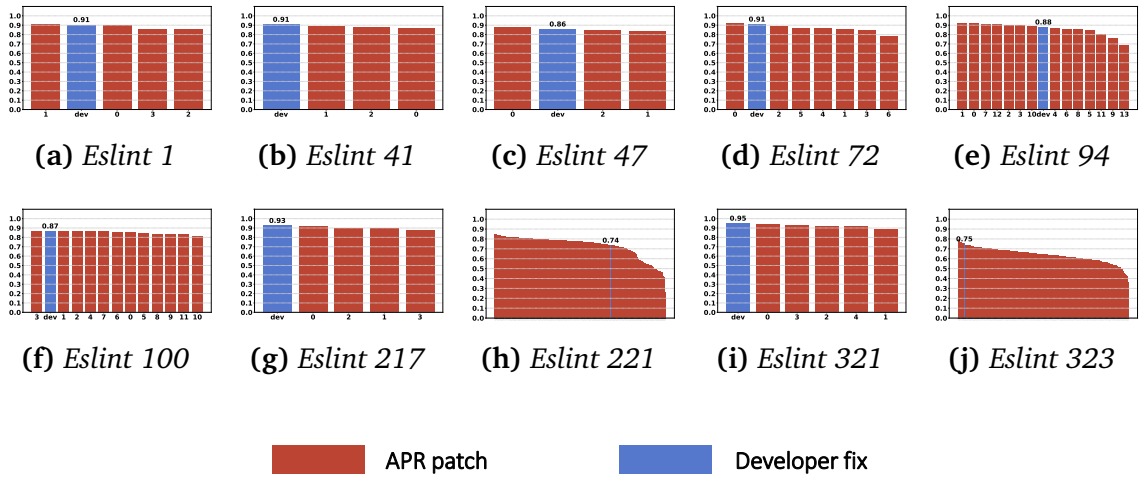


Figure 3.24: *The developer fix and patches ranked based on their similarity to the original program*

let us understand the generated line of code. The `name` variable contains a string value and it is compared with <= relation to another string value. In JavaScript if both values are strings, they are compared based on the values of the Unicode code points they contain. Meaning that every string which begins with a letter in front of `S` in alphabetical order will evaluate to `true` otherwise `false` - according to this if the `name` variable has the value `"Reflect"` or `"JSON"` it will evaluate to `true`. So far not that bad. Surprisingly changing the logical *or* (`||`) operator to the bitwise or operator (`|`) does not have any effect here, since the bitwise operation `false | true` results in `1` which converted to boolean evaluates to `true`. Similarly true for every case of the logical operation.

```
1   const name = node.callee.name;
2
3 - if (name === "Math" || name === "JSON") {
4 + if (name === "Math" | name <= 'S') {
5     context.report(node, "'{{name}}' is not a function.", { name
    });
6   }
7
```

Listing 5.1: *Original code of Eslint 1 (-), and the most similar automatically generated patch to it (+)*

Overall classifying this patch as an incorrect feels a bit ill-judged. If an experienced software developer examines this code modification, he comes to the conclusion that the fix has something to do with the `name` variable. However it might be true that the generated patch is overfitted, it contains valuable information about

the repair i.e. gives a hint to the developer which variable might cause the incorrect behaviour.

> *Based on the observed patches, a more sophisticated technique is needed to validate patches than plain source code embeddings. As we have seen the problem itself is more nuanced and complex than a simple true/false classification that can be decided using a threshold.*

One can argue that this is due to the fact that fixes are often limited to a single line, and in some cases only a single character is affected (eg. $>$ instead of $<$ in an `if` structure). It needs to be mentioned that the patches were generated with the use of a single APR tool, it is hard to justify if the conclusions are valid for other tools and multi-line fixes as well. However, defining a threshold and based on this deciding on the correctness of a patch, seems to oversimplify the decision criterion too much. On the other hand, the strive for understandable and simple patches is a reasonable and important aspect of automatic software repair. Generating unreadable patches does not help much with a real-life problem. But if a patch is not too similar to the original program, does it exclusively mean that it is unreadable? On Listing 5.2 we can observe another code snippet, but this time the least similar generated patch is picked from the bug Eslint 321. At first glance the two lines seem to be very similar even though in the similarity list it was the last one. The latter does not necessarily mean a big difference, especially if there are very few candidates: in this special case even the last plausible patch shows great similarities with the original program.

```
1 fix: fixer=>fixer.insertTextBefore(node, "\n")
2  });
3
4 - else if(tokenBefore.loc.end.line !== node.loc.end.line && option==="beside") {
5 + else if(tokenBefore.loc.end.line - node.loc.start.line && option==='beside') {
6     context.report({
7         node,
8
```

Listing 5.2: *Original code of Eslint 321 (-), and the least similar automatically generated patch to it (+)*

In case of Eslint 321 the developer fix only changed the `end` word to `start`. This is obviously a small bug and is probably due to developer inattention. Though the automatically generated fix also changed this class member it made further changes. First it changed the double quotes (`"`) to single ones (`'`). Next deleted the strict not equal operator (`!==`) to subtraction (`-`). The first change obviously did not affect the meaning of the `if` structure, but neither did the latter, because if the observed two

values are equal and if we subtract them, the result is 0, which is evaluated to `false` in JavaScript.

The last item in the similarity list can also be a semantically correct one, even though it is less similar to the original program. From this, it can be concluded that while similarity-based methods may be suitable for filtering out too many patches, one should use them for classification cautiously bearing in mind the possible misclassification.

3.5.2 Feature-based Classification

Feature Selection

To examine the effect of each feature, first 10 independent trainings were carried out and executed the feature selection algorithm described in Section 3.3.2. The results of the feature selection can be observed in Table 3.16. In the table, the feature set which opts for the best results (cells of color gray) and the intersection of the 10 independent trainings (cells of color violet) are included. In the performed experiments, we found that the output of the feature selection algorithm varies greatly due to the effect of random factors during the training phase. Despite these differences, it is clear that most features can be opted out, and that hand-crafted ones form the most important subset of such features. It is also interesting to observe that some embedding dimensions hold more valuable information than others.

Based on the experiment data, on the used ML models and parameter configurations, some features are not beneficial - omitting these does not affect the results negatively, quite the opposite, precision and recall improved in best scenarios. Overall 43 features have been selected by a single run and 15 joint features have been identified across all the 10 independent trainings.

To further investigate each feature subset, 10 independent classifiers have been trained on each. The results are listed in Table 3.17. What we can see is that the *MIN* and *MAX* values vary greatly in all of the feature sets. Despite the deviations, it is evident that some features can be opted out without any negative consequences and that on average the $\text{RFECV}_{\text{intersect}}$ yielded the best result in Precision, while $\text{RFECV}_{\text{best}}$ in Recall and F1 - thus it is more suitable for PCC. Certain embedding dimensions apear to contain valuable information; however, the model fails to encompass all necessary components. Sole reliance on embeddings led to a decrease in classifier performance. A future research direction could be to investigate what (if any) embedding dimension is equivalent to which hand-crafted/engineered feature.

The selected engineered features `rmLineNo` and `P4J_LATER_NONZERO_CONST_VF` seem

Table 3.16: *Features selected using the RFECV algorithm:*
features that yield best performance for a single execution among the 10 feature selections.
intersection between all of the features that were selected in the 10 feature optimization turns.

Hand-crafted	Engineered	Distances	Embeddings		
s3-tool	patchedFileNo	cosine_distance	vec_dim_0	vec_dim_62	vec_dim_182
AST-tool	addLineNo	braycurtis_distance	vec_dim_5	...	vec_dim_183
Cosine-tool	rmLineNo	canberra_distance	vec_dim_6	vec_dim_84	vec_dim_184
s3variable-tool	insertIfFalse	chebyshev_distance	vec_dim_7	...	vec_dim_185
variable-tool	updIfFalse	cityblock_distance	vec_dim_8	vec_dim_90	vec_dim_186
syntax-tool	ifFalse	euclidean_distance	...	...	...
semantic-tool	dupArgsInvocation	minkowski_distance	vec_dim_12	vec_dim_108	vec_dim_192
structural_score	removeNullinCond	seuclidean_distance	...	vec_dim_109	...
conceptual_score	condLogicReduce		vec_dim_21	vec_dim_110	vec_dim_200
	insertBooleanLiteral		...	vec_dim_111	...
	insertNewConstLiteral		vec_dim_27	...	vec_dim_208
	UpdateLiteral		...	vec_dim_130	vec_dim_209
	wrapsTryCatch		vec_dim_31	...	vec_dim_210
	...		vec_dim_32	vec_dim_144	vec_dim_211
	P4J_LATER_MEMBER_VF		...	vec_dim_145	...
	P4J_LATER_MODIFIED_SIMILAR_VF		vec_dim_43	...	vec_dim_220
	P4J_LATER_MODIFIED_VF		...	vec_dim_161	vec_dim_221
	P4J_LATER_NONZERO_CONST_VF		vec_dim_51	...	...
	P4J_LATER_OP_ADD_AF		vec_dim_52	vec_dim_167	vec_dim_225
	...		...	...	...
	S6_METHOD_THROWS_EXCEPTION		vec_dim_58		vec_dim_243
9 / 7 7	217 / 2 2	8 / 1 0	256 / 33 6	**Overall: 490 / 43 15**	

to grasp an important aspect of PCC, as these are selected in all feature selection attempts. Together with the hand-crafted features, these form the most essential part of the features. Table 3.17 supports this observation, as the $\text{Engineered}_{\text{plus}}$ subset yields only slightly lower F1 values than the optimized sets. However, it should be noted that Engineered features only bring an additional absolute growth of 1% on average, which can also be attributed to random factors. Random interplay is reflected in huge differences in performance in our experiments. This is not unique for PCC, but for example, if we consider the subset of the distance metrics, it can be seen that in the worst-case scenario it achieved 0% precision, while on the best case 100% precision (but on average quite moderate). We did not explore the random effects on the embedding model but hypothesize that they might have a similar impact. By selecting alternative features, not limited to embeddings and distances, one may potentially mitigate this effect.

Model Selection

Prior studies on PCC [172, 194] have exhibited a predilection for Random Forest as the classifier of choice. Additionally, it has been widely adopted in addressing various Software Engineering-related issues due to its demonstrated efficacy across diverse tasks [15, 18]. These experiences drove our intuition to use Random Forest

Table 3.17: *Measures on various feature subsets.*

		F1	Prec	Recall
MIN	All	.62	.65	.59
	$RFECV_{best}$	**.65**	**.69**	.59
	$RFECV_{intersect}$	.58	.58	.50
	Distances	.00	.00	.00
	Embeddings	.54	.51	.45
	Engineered	.56	.52	.55
	$Engineered_{plus}$	.63	.63	**.61**
	Hand-crafted	.17	.35	.10
MEAN	All	.80	.81	**.82**
	$RFECV_{best}$	**.81**	.84	.78
	$RFECV_{intersect}$	.80	**.85**	.76
	Distances	.18	.59	.11
	Embeddings	.70	.69	.71
	Engineered	.76	.76	.76
	$Engineered_{plus}$	.77	.77	.78
	Hand-crafted	.42	.57	.35
MAX	All	**.92**	.96	.96
	$RFECV_{best}$	.91	**1.00**	**.97**
	$RFECV_{intersect}$	.91	**1.00**	.90
	Distances	.41	**1.00**	.31
	Embeddings	.81	.88	.90
	Engineered	.89	.95	.93
	$Engineered_{plus}$	.90	.96	.93
	Hand-crafted	.59	.89	.61

Table 3.18: *Evaluation of the $RFECV_{best}$ feature set on 9 ML classifiers.*

		F1	Prec	Recall
MIN	DecisionTree	.46	.45	.45
	GaussianNB	.44	.30	**.82**
	KNeighbors	.45	.43	.46
	LDA	.52	.46	.54
	LogRegression	**.62**	.59	.62
	MLPClassifier	**.62**	.65	.59
	RandomForest	.61	**.70**	.48
	SGDClassifier	.55	.53	.52
	SVC	.60	.60	.52
MEAN	DecisionTree	.64	.64	.65
	GaussianNB	.51	.35	**.94**
	KNeighbors	.70	.69	.71
	LDA	.67	.60	.76
	LogRegression	.74	.69	.79
	MLPClassifier	**.80**	.81	.82
	RandomForest	.77	**.86**	.71
	SGDClassifier	.71	.68	.75
	SVC	.76	.74	.79
MAX	DecisionTree	.79	.81	.89
	GaussianNB	.56	.39	**1.00**
	KNeighbors	.83	.85	.90
	LDA	.80	.72	.97
	LogRegression	.88	.89	**1.00**
	MLPClassifier	**.92**	.96	.96
	RandomForest	.91	**1.00**	.93
	SGDClassifier	.84	.89	.93
	SVC	.89	.88	.97

*The grouped rows (*MIN, MEAN, MAX*) indicate the minimum, average and maximum values of each metric we used in the 10 independent trainings. Each subset contains the followings:* All *(all 490 features),* $RFECV_{best}$ *(43 features from the feature selection algorithm),* $RFECV_{intersect}$ *(the 15 joint feature that were selected in all 10 runs),* Distances *(the 8 distance metrics),* Embeddings *(256 dimension of the embedded code vectors),* Hand-crafted *(9 hand-crafted features from previous studies),* Engineered *(217 feature from ODS) and* $Engineered_{plus}$ *(static features comprising of hand-crafted and engineered ones).*

in feature selection, but it also raises the question of whether other classifiers might outperform it. In the subsequent experiment, 9 classifiers have been trained 10 times each to obtain the results presented in Table 3.18. The $RFECV_{best}$ feature set was used obtained in the previous section on all observed models. What we can see is that on average the MLPClassifier is the most harmonic: the F1 metric reaches highest values here on average. Also, apart from the GaussianNB classifier (which is insufficient in terms of Precision), MLP provides the highest Recall values.

While GaussianNB consistently produced the highest Recall values, its efficacy in precisely detecting overfitting patches appears inefficient. On the other hand, as previous studies suggested, RandomForest consistently provides reliable results, however, our findings indicate that MLPClassifier outperforms it by a small margin.

Relying only on the *MAX* values would flow the findings of our paper, thus we try to see the whole picture and are looking for a classifier that works well in real life scenarios most of the time (even in the worst case). What we can see in Table 3.18 is that the MLPClassifier performs reasonably well compared to other models in the *MIN* case, also. On the other hand, the motivation behind the use of Random Forest is understandable: it provides a well-explainable output with moderate training costs. The benefit of the MLPClassifier might lie in its flexibility, as the Multilayer Perceptron is a built-in model within scikit-learn with limited possibilities for customization, building a neural network from scratch and including domain-specific knowledge might add additional value to this model. While in this section every feature is treated equally and are combined naively (i.e. forming a feature vector which includes all the features of a subset), in the following we explore the possibility to combine the selected features dynamically by expanding the MLPClassifier and implementing a Neural Network in Pytorch.

Improvements of the PCC Classifier

As depicted in Figure 3.22 (b) and described in Section 3.3.2, we further tried to enhance the performance. In the previous experiments the $\text{RFECV}_{\text{best}}$ feature set was already identified as the 43 features worth training on and the MLPClassifier due to its flexible nature and reliable outcome. A Neural Network has been constructed that learns a deep representation of the input features; the measured results are shown in Table 3.19. Having a Neural Network also gives the possibility to weight input features - apart from filtering unnecessary features out this approach can give different weights to features depending on how important they are. Similarly to the previous experiments, the model was trained and evaluated 10 times, thus the *MIN*, *MEAN*, and *MAX* values are displayed of each metric. It is evident that the metric values did not improve on average (or at least not significantly, which cannot be attributed to random factors). On the other hand, the stability of the approach improved: the previous absolute deviation of 30% in the F1 score has been reduced to 16%, thus making the model much more reliable than before. These ML predictors are complementary to other state-of-the-art methods and similar to them in filtering out patches generated by APR tools (Tian *et al.* [167] 79%, Wang *et al.* [172] 87% F1 score).

Table 3.19: *Evaluation of Deep Representation Learning*

	F1	Precision	Recall
MIN	72.73%	69.70%	68.97%
MEAN	81.92%	80.77%	83.68%
MAX	88.89%	91.67%	96.55%

Ensemble Learning [143] and Majority Voting [139] are both techniques employed in ML to enhance predictive performance by combining the outputs of multiple (usually weak) individual models. Through the aggregation of predictions, majority voting leverages the collective wisdom of diverse models to make decisions. As decision in a Random Forest is obtained by majority voting of the individual trees, it alone can be treated as a Majority Voting approach. However, several recent approaches integrate the learned models either by Ensemble Learning or by Majority Voting strategies. To investigate the performance of such approaches, the combination of the nine observed ML model is also used by weighting their output. We call this method *stacking*, and it is based on a StackingRegressor [156]. Stacked generalization consists of stacking the output of individual estimator and using a regressor to compute the final prediction.

Table 3.20: *Results showcasing the stacked performance of the 9 ML models.*

	F1	Precision	Recall
Min	64.15%	66.67%	58.62%
Mean	82.38%	85.83%	79.68%
Max	93.10%	96.30%	93.10%

The results of this approach can be observed in Table 3.20. Stacking allows the useage of the strength of each individual estimator by using their output as input of a final estimator. Although the F1 score and Precision improved on average, Recall decreased making this method unsuitable for PCC. Another unfavorable inspection suggests that the deviation of all three metrics has doubled compared to the previous measurement. During the experiment, we also noticed that results are close to the ones obtained with the MLPClassifier. After further investigation we found that the algorithm assigns most of its weights to the MLP (on average 26%), Random Forest (38%) and SVC (32%) classifiers and relies only negligibly on other models. The implication of this observation suggests that machine learning models exhibit equal confusion regarding the remaining incorrectly classified samples, whereas the correctly classified examples are largely identical. The underlying reason might be data quality, inaccurate oracle (human error on classification), imbalanced data, suboptimal network architecture and parameterization, etc.

Improvement in stability can be achieved to a certain extent; however, the improvements may not suffice to ensure consistently reliable outcomes. Both Deep Representation Learning and Stacking failed at improving filtering out overfitting patches, although the former yields similar results with a more reliable standard deviation.

3.6 Discussion

3.6.1 Patch filtering based only on similarity

Natural language processing methods are widely applied in Software Engineering research, even in the APR domain. Document/sentence embedding methods were employed on source code to qualify the reliability of candidate patches. Since these methods are intended for natural language texts, first the tokenization needed to be adapted. In total 465 automatically generated patches were used in the similarity-based Patch Correctness Check study. From these 13 were syntactically equivalent to the developer fix and 211 semantically. We found that most of the semantic-matched patches were more similar to the original code than others. This behavior can be observed on Figure 3.23, where the metric values were calculated using data annotation based on the correctness of each patch. The similarity of the developer fix also tends to be close to the original program as one can see on Figure 3.24. Experiments targeted one-line modification and the evaluation was conducted on only one project. These might seem to be limitations, however at the time of writing the dissertation, there was no available APR tool, which could generate multi-line patches for JavaScript programs. As APR methods advance, one can expect that a more complex language understanding models, like Bert, would be advantageous in deciding patch correctness.

3.6.2 Using Features for Patch Classification

In this experiment, the primary emphasis lay in optimizing the features utilized within machine learning models, with a secondary focus on enhancing performance and stability through deep learning techniques. Only the domain of PCC was examined, however, this might be a general problem in Software Engineering research using ML. As we have seen, features significantly influence both the performance and stability of the applied machine learning model; however, careful construction of a neural architecture may also enhance stability. Through the application of this method, improved practices can be established for the publication of machine learning applications and the assessment of their stability in APR and also in the wider domain of Software Engineering.

An interesting insight arises from the study of Wang *et al.* [172], where they achieved 87.01% Precision and 89.14% Recall using only Hand-crafted features and a Random Forest classifier, contrasting sharply with our own results of 57% Precision and 35% Recall on average using the same features. While these figures closely approximate Wang *et al.* 's results under the best-case scenario (MAX), they remain unreproducible. Another factor to consider is the choice of library; the aforementioned article utilized the Weka app [175], which, by its graphical interface, inadvertently undermines reproducibility, unlike our use of the scikit-learn library. Reproducibility can be significantly improved by sharing a reproduction package; however, the applicability of the proposed model remains limited without evaluating its stability. Notably, the selected feature set $\text{RFECV}_{\text{best}}$ yields more reliable results than previous iterations, and the constructed neural networks contribute to the stability and reproducibility of our study. Additionally, the online appendix offers full reproducibility of the experiments conducted [5].

3.7 Concluding remarks

Patch validation in the APR domain is a less explored area, which holds great potential. Filtering out incorrect patches from the set of plausible programs is an important step forward to boost the confidence towards APR tools. In this chapter experiments were conducted both with a similarity-based patch filtering and feature-based classification approach. The similarity between patches was calculated with the use of source code embeddings produced by Doc2Vec. Although the applied approach may be useful when a high number of plausible patches are present, we found that plain source code embeddings fail to capture nuanced code semantics, thus a more sophisticated technique is needed to correctly validate patches. It is expected that a more complex language understanding model may be advantageous in deciding whether a patch is correct or not.

On the other hand, we acquired 490 features, comprising both engineered features and code embeddings, to address Patch Correctness Check. Initially, a feature selection algorithm was used to extract 43 features from the extensive feature set, indicating the limited informational contribution of most original features. Subsequently, we conducted training and evaluation of nine machine learning models to discern the optimal performer. To counteract random factors, each model underwent 10 training iterations using different random seeds. Our findings suggest better performance of the models on average when utilizing the selected feature set in comparison to the entire feature set or other subsets. Among the models examined, Multi-Layer Perceptron (MLP) and Random Forest consistently exhibited the most reliable results, achieving average F1 scores of 0.8 and 0.77, respectively. However, due to random factors, the MLP score fluctuated to 0.62 in unfavorable cases or peaked

at 0.92 in fortuitous circumstances. Employing a more complex neural architecture that integrates learned embeddings with other features enabled us to mitigate this variability, reducing the absolute fluctuation in the F1 score from 30% to 16%.

Research results underscores two major implications. First, the development of PCC classifiers requires careful planning of both feature selection and model construction; While hand-crafted features remain paramount, embeddings may also contain useful information. Second, machine learning methodologies must prioritize model stability, as it profoundly influences the validity and significance of results.

The author of this PhD thesis is responsible for the following contributions presented in this chapter:

III/1. Definition of similarity-based PCC.

III/2. Design and implementation of the Doc2Vec model and similarity measurements.

III/3. Evaluation of PCC experiments, relying on both similarities and classifiers.

III/4. Feature gathering and arrangement for classifiers.

III/5. Planning and coordinating most of the experiments.

SUMMARY

Three main topics has been discussed in this PhD thesis, which are all related to each other. The quality of a software product depends heavily on developer habits and best practices. From the thesis one can see the importance of test cases - starting from their naming, through their role in Fault Localization and APR.

Textual Similarity Techniques in Code Level Traceability

The first part of the dissertation, Chapter 1, provides insight into a traceability problem: to connect test cases with code classes solely base on textual methods. The thesis examines several prevalent methods, such as naming conventions and LSI, and introduces a fresh alternative: Doc2Vec. It experimented with different source code representations and found that IDENT, a simple representation, yielded better results for traceability. Doc2Vec-based similarity outperformed other methods. Combining Doc2Vec with recommendations from other approaches further improved performance, establishing a successful mixed approach for matching tests with production code. It is evident that a combination of methods yields optimal results in this field, and textual analysis is expected to remain important in future work.

Contributions of the thesis

In the first thesis group, the author implemented the Doc2vec and TF-IDF methods for recovering traceability links. Additionally, he implemented the text-based recovery technique that retrieved call graph information from static code. The definition of the used source code representations and metric visualizations was also part of the author's work. He also took part in the evaluation and explanation of various other results, as well as in the planning and writing of all the published papers. Detailed discussion can be found in Chapter 1.

Journal publications

[1] András Kicsi, **Viktor Csuvik** and László Vidács. Large scale evaluation of natural language processing based test-to-code traceability approaches. *IEEE Access*, Volume(9), 79089-79104, 2021.

Full papers in conference proceedings

[1] **Viktor Csuvik**, András Kicsi, and László Vidács. Evaluation of Textual Similarity Techniques in Code Level Traceability. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) - LNCS*, Springer, 529–543, 2019.

[2] **Viktor Csuvik**, András Kicsi, and László Vidács. Source code level word embeddings in aiding semantic test-to-code traceability. In *10th International Workshop at the 41st International Conference on Software Engineering (ICSE) – SST*, IEEE, 29-33, 2019.

Machine Learning in Automated Program Repair

Chapter 2 started with a discussion on Fault Localization using Deep Learning. The findings of the study can be generalized to the whole SE and AI domain: scientific work using ML should concentrate more on reproducibility and stability aside from publishing great results. Next, a data mining approach has been presented, and the FixJS dataset. It can be used for APR research: just as in the subsequent parts where patches are generated both by DL and traditional approaches. Results show that in practical application G&V APR approaches still play a prominent role, with DL-based tools outperforming them in some cases. An important and difficult task for future research will be to combine the strengths of the two areas and avoid the weaknesses. The dissertation presents a thorough examination of the effectiveness of genetic operators and showcased instances of potential patches discovered by both of the algorithms. Offering both this study and its associated GitHub repository, we aim to streamline APR research and aspire for it to become a foundational resource for future endeavors concerning JavaScript programs.

Contributions of the thesis

In the second thesis group, the author coordinated the experimentations on diverse network architectures on DL-based FL and implemented the bucketing approach. He also adapted churn metric and took part in the design and writing of the published paper. The FixJS benchmark creation and ChatGPT experiments were entirely the work of the author. In the GenProgJS tool, the author implemented the base genetic algorithm, and the interface for test case evaluation and operator calls. He also executed the experiments, coordinated the analysis and took a big part in the explanation of results. Detailed discussion can be found in Chapter 2.

Full papers in conference proceedings

[1] **Viktor Csuvik**, Roland Aszmann, Árpád Beszédes, Ferenc Horváth, and Tibor Gyimóthy On the stability and applicability of deep learning in fault localization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2024.

[2] **Viktor Csuvik**, and László Vidács. Fixjs: A dataset of bug-fixing javascript commits. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, ACM, 712–716, 2022.

[3] **Viktor Csuvik**, Tibor Gyimóthy, and László Vidács. Can chatgpt fix my code?. In *Proceedings of the 18th International Conference on Software Technologies - ICSOFT*, SciTePress, 478-485 2023.

Automated Assessment of Automatically Generated Patches

The third part tried to tackle with the PCC problem - that is, given an automatically generated patch, one should decide whether it is a real fix to the bug, or an overfit to the test oracle. Chapter 3 elaborates on how our work contributed to the field, by defining similarity-based patch filtering and evaluating classification on state-of-the-art features sets. In the realm of APR, patch validation remains relatively uncharted yet promising. Filtering out erroneous patches is crucial for enhancing confidence in automatic tools. This chapter explores experiments employing both similarity-based patch filtering and feature-based classification methods. Results show that while the current solutions still have some flaws, by selecting proper features and classifiers, one can filter out overfitting patches with a high degree of confidence. The research findings emphasize two key points. Firstly, constructing effective PCC classifiers demands meticulous consideration of feature selection and model construction. While handcrafted features remain essential, embeddings can also offer valuable insights. Secondly, ML approaches should prioritize model stability, as it significantly impacts the reliability and importance of the results obtained - a similar observation made in previous section.

Contributions of the thesis

In the third thesis group, the author laid the groundwork for the similarity-based PCC technique and implemented the base algorithm. He took part in the manual annotation of the generated patches. The author coordinated the implementation of the ML-based classifiers and conducted benchmark creation / gathering of all required

metrics. He also planned the experiment guidelines and took a big role in the evaluation and explanation of the results and their implications. Detailed discussion can be found in Chapter 3.

Full papers in conference proceedings

[1] **Viktor Csuvik**, Dániel Horváth, Ferenc Horváth, and László Vidács. Utilizing Source Code Embeddings to Identify Correct Patches. In *IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, IEEE, 18–25, 2020.

[2] **Viktor Csuvik**, Dániel Horváth, Márk Lajkó, and László Vidács. Exploring plausible patches using source code embeddings in javascript. In *IEEE/ACM International Workshop on Automated Program Repair (APR)*, IEEE, 11–18, 2021.

[3] **Viktor Csuvik**, Dániel Horváth, and László Vidács. Feature extraction, learning and selection in support of patch correctness assessment. In *Proceedings of the 19th International Conference on Software Technologies - ICSOFT*, SciTePress, 2024.

Acknowledgement

I am incredibly thankful for the support of my family during my academic journey, as well as other individuals. While it is impossible to name everyone, I want to express my deepest gratitude to my supervisor, László Vidács, for his exceptional mentorship. I am also grateful to my co-authors, colleagues, including András Kicsi, Dániel Horváth, Márk Lajkó and Ferenc Horváth who have played pivotal roles in our research success. Special thanks to Tibor Gyimóthy for offering me a doctoral position and providing enriching research opportunities.

The research presented in this dissertation was supported in part by the ÚNKP-19-2-SZTE-19, ÚNKP-20-3-SZTE-457, ÚNKP-21-3-SZTE-385, ÚNKP-22-3-SZTE-396 and ÚNKP-23-3-SZTE-435 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund, and by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory. The national project TKP2021-NVA-09 also supported this work. Project no TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

ÖSSZEFOGLALÁS

A doktori disszertáció három fő témát tárgyal, amelyek mindegyike valamelyest kapcsolódik egymáshoz. Egy szoftvertermék minősége nagyban függ a fejlesztői szokásoktól és gyakorlatoktól. Az értekezésből látható a tesztesetek fontossága - kezdve a megnevezésüktől a hibalokalizációban és az automatikus programjavításban betöltött szerepükben.

Szöveges Hasonlósági Technikák a Kódszintű Nyomonkövethetőségben

A disszertáció első része, a fejezet, betekintést nyújt egy nyomonkövethetőségi problémába: a tesztesetek összekapcsolásába kódosztályokkal, kizárólag szöveges módszerek felhasználásával. A dolgozat számos jól ismert módszert vizsgál, mint például a névkonvenciókat, az LSI-t, és bemutat egy új alternatívát is: a Doc2Vec-et. Különböző forráskód reprezentációk kerültek bemutatásra, és azt láthattuk, hogy az egyszerű IDENT reprezentáció jobb eredményeket adott a nyomonkövethetőség tekintetében a többi szöveges reprezentációtól. A Doc2Vec-alapú hasonlóság felülmúlta a többi módszert. A Doc2Vec más megközelítésekből származó hasonlósági listákkal való kombinálása tovább javította a teljesítményt, és sikeres kombinált megközelítést hozott létre a tesztek és az osztályok összekapcsolására. Láthattuk, hogy a szöveges technikák rugalmas megközelítést biztosítanak, valamint ezek kombinációja javítja a teljesítményt, így a szövegelemzés várhatóan továbbra is fontos marad a jövőbeni munkákban.

A disszertáció hozzájárulásai

Az első téziscsoportban a szerző a Doc2vec és a TF-IDF módszereket valósította meg a teszt-kód kapcsolatok helyreállítására. Emellett megvalósította a szövegalapú helyreállítási technikát, amely statikus kódból hívásgráf-információkat (Call Graph - CG) nyert vissza. Az alkalmazott forráskód-reprezentációk és metrikus vizualizációk meghatározása szintén a szerző munkájának részét képezte. Részt vett továbbá a legtöbb eredmény kiértékelésében és magyarázatában, valamint az összes publikált cikk megírásában. Részletes tárgyalása az fejezetben található.

Folyóirat publikációk

[1] András Kicsi, **Viktor Csuvik** and László Vidács. Large scale evaluation of natural language processing based test-to-code traceability approaches. *IEEE Access*, Volume(9), 79089-79104, 2021.

Konferenciakötetben megjelent teljes publikációk

[1] **Viktor Csuvik**, András Kicsi, and László Vidács. Evaluation of Textual Similarity Techniques in Code Level Traceability. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) - LNCS*, Springer, 529–543, 2019.

[2] **Viktor Csuvik**, András Kicsi, and László Vidács. Source code level word embeddings in aiding semantic test-to-code traceability. In *10th International Workshop at the 41st International Conference on Software Engineering (ICSE) – SST*, IEEE, 29-33, 2019.

Gépi Tanulás az Automatikus Programjavítás Területén

A 1.9 fejezet a mélytanulás alapú hibalokalizáció tárgyalásával kezdődik. A fejezet megállapításai általánosíthatók az egész szoftverfejlesztés és mesterséges intelligencia területre: a gépi tanulást használó tudományos munkáknak a nagyszerű eredmények publikálása mellett jobban kellene koncentrálnia a reprodukálhatóságra és a stabilitásra. Ezután egy adat kinyerési megközelítés került bemutatásra, és a FixJS adathalmaz. Ennek egyik alapvető felhasználása az automatikus programjavítás kutatása: akárcsak ahogy ez a következő fejezetben bemutatásra is kerül, ahol a patchek mind mélytanulással, mind hagyományos megközelítésekkel generálódnak. Ezek az eredmények azt mutatják, hogy a gyakorlati alkalmazásban a hagyományos genetikus megközelítések még mindig kiemelkedő szerepet játszanak, a mélytanuló eszközök csak egyes esetekben múlják felül azokat. A jövőbeli kutatások fontos és nehéz feladata lesz e két terület erősségeinek ötvözése a gyengeségek elkerülésével. A disszertáció alaposan megvizsgálja a genetikus operátorok hatékonyságát, és bemutatja a mindkét algoritmus által generált potenciálisan javító patch-eket. A cikket és a hozzá kapcsolódó GitHub repository-t egyaránt publikussá téve célunk az automatikus programjavítás kutatás fejlesztése, és arra törekszünk, hogy a JavaScript programokkal kapcsolatos jövőbeli törekvések alapvető kiindulópontjává váljon.

A disszertáció hozzájárulásai

A második téziscsoportban a szerző koordinálta a különböző neurális hálózati architektúrákon végzett kísérleteket a mélytanulás alapú hibalokalizációt kutatva, és

megvalósította a bucketing (dobozolás) megközelítést. Emellett adaptálta a churn metrikát, és részt vett a publikált cikk megtervezésében és megírásában. A FixJS adathalmaz létrehozása és a ChatGPT kísérletek teljes egészében a szerző munkái. A GenProgJS eszközben a szerző implementálta az alap genetikus algoritmust, valamint a tesztesetek kiértékeléséhez és az operátorhívásokhoz szükséges interfészt. A program futtatásokat is ő hajtotta végre, koordinálta az elemzést, és nagy szerepet vállalt az eredmények magyarázatában. Részletes értekezés a 1.9 fejezetben található.

Konferenciakötetben megjelent teljes publikációk

[1] **Viktor Csuvik**, Roland Aszmann, Árpád Beszédes, Ferenc Horváth, and Tibor Gyimóthy On the stability and applicability of deep learning in fault localization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2024.

[2] **Viktor Csuvik**, and László Vidács. Fixjs: A dataset of bug-fixing javascript commits. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, ACM, 712–716, 2022.

[3] **Viktor Csuvik**, Tibor Gyimóthy, and László Vidács. Can chatgpt fix my code?. In *Proceedings of the 18th International Conference on Software Technologies - ICSOFT*, SciTePress, 478-485 2023.

Automatikusan Generált Javítások Spontán Értékelése

A harmadik rész az automatikus patch kiértékelés problémával foglalkozik - vagyis egy automatikusan generált javítás esetén el kell dönteni, hogy az a hiba valódi javítását jelenti-e, vagy csupán túlillesztés történt a tesztekre. A 2.9 fejezet részletezi, hogy munkánk hogyan járult hozzá a területhez - a hasonlóság-alapú patch szűrés definiálásával és az osztályozók kiértékelésével a legkorszerűbb jellemzőkészleteken. Az automatikus programjavítás területén a patch-ek helyességének vizsgálata még viszonylag feltérképezetlen, mégis ígéretes terület. A hibás javítások kiszűrése kulcsfontosságú az ilyen automatikus eszközökbe vetett bizalom növeléséhez. Ez a fejezet olyan kísérleteket mutat be, amelyekben hasonlóság-alapú patch szűrést és jellemző-alapú osztályozási módszereket alkalmaztunk. Az eredmények azt mutatják, hogy bár a jelenlegi megközelítések még korán sem tökéletesek, a megfelelő jellemzők és osztályozók kiválasztásával nagyfokú megbízhatósággal ki lehet szűrni a túlillesztett javításokat. A kutatási eredmények két kulcsfontosságú következtetést mutatnak. Először is, jó osztályozók készítése érdekében a jellemzők kiválasztásánál és a modellépítésnél különös figyelmet kell szentelni a hatékonyságra. Bár a kézzel készített jellemzők továbbra is alapvető fontosságúak, a beágyazások is értékes információt

nyújthatnak. Másodszor pedig, az ilyen megközelítések alkalmazásakor prioritást kell adni a modell stabilitásának, mivel ez jelentősen befolyásolja a kapott eredmények megbízhatóságát - ez az előző fejezetben tett megállapításhoz hasonló.

A disszertáció hozzájárulásai

A harmadik téziscsoportban a szerző lefektette a hasonlóságon alapuló patch helyesség vizsgálat technika alapjait, és megvalósította az alapalgoritmust. Részt vett a generált javítások kézi elemzésében és annotálásában. A szerző koordinálta a gépi tanuló osztályozók implementálását, és elvégezte az összes szükséges metrika létrehozását és gyűjtését. Ő tervezte meg a kísérleti irányelveket is, és nagy szerepet vállalt az eredmények és azok következményeinek kiértékelésében, magyarázatában és publikálásában. Részletes leírás a 2.9 fejezetben található.

Konferenciakötetben megjelent teljes publikációk

[1] **Viktor Csuvik**, Dániel Horváth, Ferenc Horváth, and László Vidács. Utilizing Source Code Embeddings to Identify Correct Patches. In *IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, IEEE, 18–25, 2020.

[2] **Viktor Csuvik**, Dániel Horváth, Márk Lajkó, and László Vidács. Exploring plausible patches using source code embeddings in javascript. In *IEEE/ACM International Workshop on Automated Program Repair (APR)*, IEEE, 11–18, 2021.

[3] **Viktor Csuvik**, Dániel Horváth, and László Vidács. Feature extraction, learning and selection in support of patch correctness assessment. In *Proceedings of the 19th International Conference on Software Technologies - ICSOFT*, SciTePress, 2024.

Köszönetnyilvánítás

Hihetetlenül hálás vagyok a családom támogatásáért a tanulmányaim során, valamint minden barátom és ismerősöm bátorításáért. Óvodás korom óta számos nevelő és tanító járult hozzá ahhoz ami most vagyok, a jelen disszertáció az ő érdemük is. Bár lehetetlen mindenkit megnevezni, szeretném kifejezni legmélyebb hálámat témavezetőmnek, Vidács Lászlónak kivételes mentorálásáért. Hálás vagyok továbbá társszerzőimnek, kollégáimnak, köztük Kicsi Andrásnak, Horváth Dánielnek, Lajkó Márknak és Horváth Ferencnek, akik nagy szerepet játszottak a kutatási sikerekben. Külön köszönöm Gyimóthy Tibornak, hogy doktori hallgatói helyet ajánlott fel számomra és ösztönző kutatási lehetőséget biztosított.

A disszertáció részben az Innovációs és Technológiai Minisztérium ÚNKP-19-2-SZTE-19, ÚNKP-20-3-SZTE-457, ÚNKP-21-3-SZTE-385, ÚNKP-22-3-SZTE-396, ÚNKP-23-3-SZTE-435 kódszámú Új Nemzeti Kiválóság Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült, valamint az Európai Unió RRF-2.3.1-21-2022-00004 azonosítójú, Mesterséges Intelligencia Nemzeti Laboratórium projekt keretében. A TKP2021-NVA-09 nemzeti projekt szintén támogatta a munkát. A TKP2021-NVA-09 számú projekt a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból a Nemzeti Kulturális és Innovációs Minisztérium által a TKP2021-NVA támogatási séma keretében finanszírozott támogatással valósult meg.

Bibliography

[1] Gensim gensim webpage. https://radimrehurek.com/gensim/. Accessed: 2019.

[2] TIOBE programming community index. https://www.tiobe.com/tiobe-index. Accessed: 2019.

[3] Grammar-Based Patches Generation for Automated Program Repair. *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1300–1305, 2021.

[4] Supplemental material for on the stability and applicability of deep learning in fault localization, 2023.

[5] Supplemental material for "feature extraction, learning and selection in support of patch correctness assessment". https://anonymous.4open.science/r/PCC-2024-45CF/, 2024.

[6] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, 2007.

[7] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified Pre-training for Program Understanding and Generation. pages 2655–2668, mar 2021.

[8] Ilya Sutskever Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei. [GPT-2] Language Models are Unsupervised Multitask Learners. *OpenAI Blog*, 1(May):1–7, 2020.

[9] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, oct 2002.

[10] Apache Commons webpage. http://commons.apache.org/, 2019.

[11] Fatmah Yousef Assiri and James M. Bieman. Fault localization for automated program repair: effectiveness, performance, repair correctness. *Software Quality Journal*, 25(1):171–199, mar 2017.

[12] Mohammad Mahdi Bejani and Mehdi Ghatee. A systematic review on overfitting control in shallow and deep neural networks. *Artificial Intelligence Review*, pages 1–48, 2021.

[13] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[14] Srinadh Bhojanapalli, Kimberly Wilber, Andreas Veit, Ankit Singh Rawat, Seungyeon Kim, Aditya Krishna Menon, and Sanjiv Kumar. On the reproducibility of neural network predictions. *ArXiv*, abs/2102.03349, 2021.

[15] Peter Bludau and Alexander Pretschner. Feature sets in just-in-time defect prediction: an empirical evaluation. *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022.

[16] Markus Borg, Per Runeson, and Anders Ardö. Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. *Empirical Software Engineering*, 19(6):1565–1616, dec 2014.

[17] Philipp Bouillon, Jens Klinke, Nils Meyer, and Friedrich Steimann. EZUNIT: A framework for associating failed unit tests with potential programming errors. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4536 LNCS, pages 101–104. Springer Verlag, 2007.

[18] David Bowes, Tracy Hall, and Jean Petrić. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, 26(2):525–552, jun 2018.

[19] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020.

[20] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. Automatic recovery from runtime failures. *Proceedings - International Conference on Software Engineering*, pages 782–791, 2013.

[21] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic Workarounds for Web Applications. In *ACM Transactions on Software Engineering and Methodology*, volume 24, pages 1–42, New York, New York, USA, 2015. ACM Press.

[22] Chatgpt: Understanding the chatgpt ai chatbot. `https://www.eweek.com/big-data-and-analytics/chatgpt/`, 2023.

[23] Liushan Chen, Yu Pei, and Carlo A. Furia. Contract-based program repair without the contracts. Technical report, 2017.

[24] Zimin Chen, Steve James Kommrusch, and Martin Monperrus. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering*, apr 2022.

[25] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering*, (01):1–1, sep 2019.

[26] William Jay Conover. *Practical nonparametric statistics*, volume 350. John Wiley & Sons, 1998.

[27] Q. Cormier, M. Milani Fard, K. Canini, and M. R. Gupta. Launch and iterate: Reducing prediction churn. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 3179–3187, Red Hook, NY, USA, 2016. Curran Associates Inc.

[28] Viktor Csuvik., Tibor Gyimóthy., and László Vidács. Can chatgpt fix my code? In *Proceedings of the 18th International Conference on Software Technologies - ICSOFT*, pages 478–485. INSTICC, SciTePress, 2023.

[29] Viktor Csuvik, Dániel Horváth, Márk Lajkó, and László Vidács. Exploring plausible patches using source code embeddings in javascript. *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 11–18, 2021.

[30] Viktor Csuvik, Daniel Horvath, and Laszlo Vidacs. Feature extraction, learning and selection in support of patch correctness assessment. In *Proceedings of the 19th International Conference on Software Technologies - ICSOFT*, 2024.

[31] Viktor Csuvik, Deniel Horvath, Ferenc Horvath, and Laszlo Vidacs. Utilizing Source Code Embeddings to Identify Correct Patches. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 18–25. IEEE, 2020.

[32] Viktor Csuvik, András Kicsi, and László Vidács. Evaluation of Textual Similarity Techniques in Code Level Traceability. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11622 LNCS, pages 529–543. Springer Verlag, 2019.

[33] Viktor Csuvik, András Kicsi, and László Vidács. Source code level word embeddings in aiding semantic test-to-code traceability. In *10th International Workshop at the 41st International Conference on Software Engineering (ICSE) – SST 2019*. IEEE, 2019.

[34] Viktor Csuvik, Aszmann Roland, Beszédes Árpád, Horváth Ferenc, and Gyimóthy Tibor. On the stability and applicability of deep learning in fault localization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024.

[35] Viktor Csuvik and László Vidács. Fixjs: A dataset of bug-fixing javascript commits. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 712–716, 2022.

[36] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 47(1):67–85, 2021.

[37] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. QLOSE: Program repair with quantitative objectives. Technical report, 2016.

[38] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation*, pages 65–74, 2010.

[39] Ralph A. DeFronzo, Andrew Lewin, Sanjay Patel, Dacheng Liu, Renee Kaste, Hans J. Woerle, and Uli C. Broedl. Combination of empagliflozin and linagliptin as second-line therapy in subjects with type 2 diabetes inadequately controlled on metformin. *Diabetes Care*, 38(3):384–393, jul 2015.

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. oct 2018.

[41] Elizabeth Dinella, Hanjun Dai, Google Brain, Ziyang Li, Mayur Naik, Le Song, Georgia Tech, and Ke Wang. Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs. Technical report, 2020.

[42] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. Generating bug-fixes using pretrained transformers. *MAPS 2021 - Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, co-located with PLDI 2021*, pages 1–8, jun 2021.

[43] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 302–313, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Katherine Elkins and Jon Chun. Can GPT-3 Pass a Writer's Turing Test? *Journal of Cultural Analytics*, sep 2020.

[45] Esprima official website. `https://esprima.org`, 2023.

[46] Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony, and Ra'fat Al-msie'deen. Recovering traceability links between feature models and source code of product variants. In *VARiability for You Workshop on Variability Modeling Made Useful for Everyone - VARY '12*, pages 21–25. ACM Press, 2012.

[47] Xianmei Fang, Xiaobo Gao, Yuting Wang, Zhouyu Liao, and Yue Ma. Improving fault localization using conditional variational autoencoder. *IEICE TRANSACTIONS on Information and Systems*, 105(8):1490–1494, 2022.

[48] Amin Milani Fard and Ali Mesbah. Javascript: The (un)covered parts. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 230–240. IEEE, 2017.

[49] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. *IEEE International Conference on Software Maintenance, ICSM*, pages 23–32, 2003.

[50] J. M. Florez. Automated fine-grained requirements-to-code traceability link recovery. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 222–225, 2019.

[51] Alcides Fonseca and Máximo Oliveira. Figra: Evaluating a larger search space for cardumen in automatic program repair. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 24–30, 2022.

[52] Mariani Leonardo Gazzola Luca, Micucci Daniela. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, jan 2019.

[53] Online appendix for genprogjs. `https://github.com/GenProgJS/GenProgJS`, 2024.

[54] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. Automatically identifying focal methods under test in unit test cases. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 61–70. IEEE, sep 2015.

[55] Gh archive official website. `https://www.gharchive.org`, 2023.

[56] The 2023 state of the octoverse. `https://octoverse.github.com`, 2024.

[57] Github rest api official website. `https://docs.github.com/en/rest`, 2023.

[58] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, November 2019.

[59] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[60] Odd Erik Gundersen, Kevin L. Coakley, and Christine R. Kirkpatrick. Sources of irreproducibility in machine learning: A review. *CoRR*, abs/2204.07610, 2022.

[61] J. Guo, J. Cheng, and J. Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14, 2017.

[62] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 3–14. IEEE, may 2017.

[63] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 1345–1351. AAAI Press, 2017.

[64] Peter Gyimesi, Bela Vancsics, Andrea Stocco, Davood Mazinanian, Arpad Beszedes, Rudolf Ferenc, and Ali Mesbah. BugsJS: A benchmark of javascript bugs. In *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pages 90–101, apr 2019.

[65] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. Discovering bug patterns in JavaScript. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 144–156, 2016.

[66] Masum Hasan, Kazi Sajeed Mehrab, Wasi Uddin Ahmad, and Rifat Shahriyar. Text2App: A Framework for Creating Android Apps from Text Descriptions. 2021.

[67] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, Andre Van Hoorn, Antonio Filieri, and David Lo. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience*, 49(8):1197–1224, 2019.

[68] T. Hey. Indirect: Intent-driven requirements-to-code traceability. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 190–191, 2019.

[69] Djoerd Hiemstra. A probabilistic justification for using tf - idf term weighting in information retrieval. *International Journal on Digital Libraries*, 3(2):131–139, aug 2000.

[70] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. Technical report.

[71] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 34–43, New York, NY, USA, 2007. Association for Computing Machinery.

[72] Jian Hu, Huan Xie, Yan Lei, and Ke Yu. A light-weight data augmentation method for fault localization. *Information and Software Technology*, 157:107148, 2023.

[73] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 12–23, New York, NY, USA, may 2018. Association for Computing Machinery (ACM).

[74] Chen Huo and James Clause. Interpreting Coverage Information Using Direct and Indirect Coverage. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 234–243. IEEE, apr 2016.

[75] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. Deep transfer bug localization. *IEEE Transactions on software engineering*, 47(7):1368–1380, 2019.

[76] Elkhan Ismayilzada, Md Mazba Ur Rahman, Dongsun Kim, and Jooyong Yi. Poracle: Testing patches under preservation conditions to combat the overfitting problem of program repair. *ACM Trans. Softw. Eng. Methodol.*, 33(2), dec 2023.

[77] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 34–44, New York, NY, USA, 2012. Association for Computing Machinery.

[78] Heinrich Jiang, Harikrishna Narasimhan, Dara Bahri, Andrew Cotter, and Afshin Rostamizadeh. Churn reduction via distillation. *arXiv preprint arXiv:2106.02654*, 2021.

[79] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*, 18:298–309, 2018.

[80] Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: Code-aware neural machine translation for automatic program repair, 2021.

[81] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pages 437–440. Association for Computing Machinery, Inc, jul 2014.

[82] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[83] Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman. Evaluating automatic program repair capabilities to repair api misuses. *IEEE Transactions on Software Engineering*, 48(7):2658–2679, 2022.

[84] András Kicsi, László Tóth, and László Vidács. Exploring the benefits of utilizing conceptual information in test-to-code traceability. *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 8–14, 2018.

[85] András Kicsi, László Vidács, and Tibor Gyimothy. Testroutes: A manually curated method level dataset for test-to-code traceability. In *Proceedings of the 17th International Conference on Mining Software Repositories, MSR 2020*, pages 593–597. IEEE, IEEE, jun 2020.

[86] András Kicsi, Viktor Csuvik, and László Vidács. Large scale evaluation of natural language processing based test-to-code traceability approaches. *IEEE Access*, 9:79089–79104, 2021.

[87] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings - International Conference on Software Engineering*, pages 802–811. IEEE, may 2013.

[88] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. *Proceedings - International Conference on Software Engineering*, pages 150–162, may 2021.

[89] Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Trans. Softw. Eng. Methodol.*, 28(4), oct 2019.

[90] Max Klabunde, Tobias Schumacher, Markus Strohmaier, and Florian Lemmerich. Similarity of neural network models: A survey of functional and representational measures. *arXiv preprint arXiv:2305.06329*, 2023.

[91] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, pages 165–176, New York, New York, USA, 2016. ACM Press.

[92] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25(3):1980–2024, may 2020.

[93] Dinh Xuan Bach Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. On Reliability of Patch Correctness Assessment. In *Proceedings - International Conference on Software Engineering*, volume 2019-May, pages 524–535. IEEE Computer Society, may 2019.

[94] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 593–604, New York, NY, USA, 2017. Association for Computing Machinery.

[95] Xuan Bach D. Le, David Lo, and Claire Le Goues. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 213–224. IEEE, mar 2016.

[96] Xuan Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23(5):3007–3033, oct 2018.

[97] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings - International Conference on Software Engineering*, pages 3–13. IEEE, jun 2012.

[98] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, January 2012.

[99] Williams Lefebvre-Ulrikson, G. Da Costa, L. Rigutti, and I. Blum. *Data Mining*. New York, 2016.

[100] Yan Lei, Chunyan Liu, Huan Xie, Sheng Huang, Meng Yan, and Zhou Xu. Bcl-fl: A data augmentation approach with between-class learning for fault localization. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 289–300. IEEE, 2022.

[101] Yan Lei, Tiantian Wen, Huan Xie, Lingfeng Fu, Chunyan Liu, Lei Xu, and Hongxia Sun. Mitigating the effect of class imbalance in fault localization using context-aware generative adversarial network. *arXiv preprint arXiv:2303.06644*, 2023.

[102] Omer Levy and Yoav Goldberg. Linguistic Regularities in Sparse and Explicit Word Representations. Technical report, 2014.

[103] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019.

[104] Yi Li, Shaohua Wang, and Tien Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 661–673. IEEE, 2021.

[105] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. Context-aware code change embedding for better patch correctness assessment. *ACM Trans. Softw. Eng. Methodol.*, 31(3), may 2022.

[106] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2017, page 55–56, New York, NY, USA, 2017. Association for Computing Machinery.

[107] Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. Deep Learning Based Program Generation from Requirements Text: Are We There Yet? *IEEE Transactions on Software Engineering*, pages 1–1, aug 2020.

[108] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, jan 2021.

[109] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, pages 298–312, 2016.

[110] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021.

[111] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *undefined*, 2021.

[112] Thibaud Lutellier, Lawrence Pang, Viet Hung Pham, Moshi Wei, and Lin Tan. ENCORE: Ensemble Learning using Convolution Neural Machine Translation for Automatic Program Repair. 2019.

[113] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 20:101–114, 2020.

[114] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 215–226. IEEE Computer Society, 2018.

[115] Andrian Marcus, Jonathan I Maletic, and Andrey Sergeyev. Recovery of Traceability Links between Software Documentation and Source Code. *International Journal of Software Engineering and Knowledge Engineering*, pages 811–836, 2005.

[116] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, aug 2017.

[117] Matias Martinez and Martin Monperrus. ASTOR: A program repair library for Java (Demo). *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444, 2016.

[118] Matias Martinez and Martin Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of Astor. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11036 LNCS:65–86, dec 2018.

[119] Matias Martinez and Martin Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *Journal of Systems and Software*, 151:65–80, feb 2019.

[120] Nayrolles Mathieu and Abdelwahab Hamou-Lhadj. Word embeddings for the software engineering domain. *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, pages 38–41, 2018.

[121] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 448–458. IEEE, may 2015.

[122] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.

[123] Tomas Mikolov, Ilya Sutskever, Kan Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *NIPS'13 Proceedings of the 26th International Conference on Neural Information Processing Systems*, 2:3111–3119, dec 2013.

[124] Amr Mansour Mohsen, Hesham Hassan, Ramadan Moawad, and Soha H. Makady. A review on software bug localization techniques using a motivational example. *International Journal of Advanced Computer Science and Applications*, 2022.

[125] Mondrian webpage. `http://www.theusrus.de/Mondrian/`, 2019.

[126] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 234–242, New York, New York, USA, 2014. ACM Press.

[127] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), January 2018.

[128] Martin Monperrus. The Living Review on Automated Program Repair. Technical report, dec 2020.

[129] Miguel Morin and Matthew Willetts. Non-determinism in tensorflow resnets. *ArXiv*, abs/2001.11396, 2020.

[130] Marjane Namavar, Noor Nashid, and Ali Mesbah. A controlled experiment of different code representations for learning-based program repair. *Empirical Software Engineering*, 27(7), dec 2022.

[131] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Benjamin Zorn. Javascript errors in the wild: An empirical study. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering*, ISSRE '11, page 100–109, USA, 2011. IEEE Computer Society.

[132] OpenAI. Gpt-4 technical report, 2023.

[133] Openai chatgpt. `https://openai.com/blog/chatgpt/`, 2023.

[134] Openai chatgpt app. `https://chat.openai.com/`, 2024.

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

[136] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia. When and How Using Structural Information to Improve IR-Based Traceability Recovery. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 199–208. IEEE, mar 2013.

[137] Reza Meimandi Parizi, Sai Peck Lee, and Mohammad Dabbagh. Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts. *IEEE Transactions on Reliability*, 63:913–926, 2014.

[138] 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.

[139] L. S. Penrose. The elementary statistics of majority voting. *Journal of the Royal Statistical Society*, 109(1):53–57, 1946.

[140] Christian S. Perone, Roberto Silveira, and Thomas S. Paula. Evaluation of sentence embeddings in downstream and linguistic probing tasks. Technical report, 2018.

[141] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. CoTexT: Multi-task Learning with Code-Text Transformer. pages 40–47, may 2021.

[142] Quang-Ngoc Phung, Misoo Kim, and Eunseok Lee. Identifying incorrect patches in program repair based on meaning of source code. *IEEE Access*, 10:12012–12030, 2022.

[143] Robi Polikar. Ensemble learning. *Ensemble machine learning: Methods and applications*, pages 1–34, 2012.

[144] Julian Aron Prenner, Hlib Babii, and Romain Robbes. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. *Proceedings - International Workshop on Automated Program Repair, APR 2022*, pages 69–75, 2022.

[145] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 24–36, New York, NY, USA, 2015. Association for Computing Machinery.

[146] Jie Qian, Xiaolin Ju, and Xiang Chen. Gnet4fl: Effective fault localization via graph convolutional neural network. *Automated Software Engg.*, 30(2), apr 2023.

[147] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software*, 88:147–168, 2014.

[148] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. Evaluating test-to-code traceability recovery methods through controlled experiments. *Journal of Software: Evolution and Process*, 25(11):1167–1191, nov 2013.

[149] Zhijun Ren, Tantao Lin, Ke Feng, Yongsheng Zhu, Zheng Liu, and Ke Yan. A systematic review on imbalanced learning methods in intelligent fault diagnosis. *IEEE Transactions on Instrumentation and Measurement*, 2023.

[150] Rfecv documentation. `https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html`, 2024.

[151] Diego Rodríguez-Baquero and Mario Linares-Vásquez. Mutode: Generic JavaScript and node.js mutation testing tool. In *ISSTA 2018 - Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 372–375, New York, New York, USA, jul 2018. Association for Computing Machinery, Inc.

[152] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In *European Conference on Software Maintenance and Reengineering, CSMR*, pages 209–218. IEEE, 2009.

[153] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. Elixir: Effective object-oriented program repair. In *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659. Institute of Electrical and Electronics Engineers Inc., nov 2017.

[154] Susanta Sarangi, Md Sahidullah, and Goutam Saha. Optimization of data-driven filterbank for automatic speaker verification. *Digital Signal Processing*, 104:102795, 2020.

[155] Scikit-learn documentation. `https://scikit-learn.org/stable/user_guide.html`, 2024.

[156] Scikit-learn stackingregressor. `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingRegressor.html`, 2024.

[157] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[158] M. Selakovic and M. Pradel. Poster: Automatically fixing real-world javascript performance bugs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 811–812, 2015.

[159] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 532–543, New York, New York, USA, 2015. ACM Press.

[160] Victor Sobreira, Thomas Durieux, Fernanda Madeiral Delfim, Monperrus Martin, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140, 2018.

[161] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283, 2017.

[162] Stack overflow developer survey results 2023. `https://insights.stackoverflow.com/survey/2023`, 2024.

[163] Senthil Karthikeyan Sundaram, Jane Huffman Hayes, and Alexander Dekhtyar. Baselines in requirements tracing. In *ACM SIGSOFT Software Engineering Notes*, volume 30, page 1, New York, New York, USA, 2005. ACM Press.

[164] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 727–738, New York, New York, USA, 2016. ACM Press.

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

[166] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kaboré, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. Predicting patch correctness based on the similarity of failing test cases. *ACM Trans. Softw. Eng. Methodol.*, 31(4), aug 2022.

[167] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F. Bissyandé. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. *ACM Trans. Softw. Eng. Methodol.*, 32(4), may 2023.

[168] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 25–36. IEEE Press, 2019.

[169] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, 18:542–553, 2018.

[170] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2023.

[171] Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. How Different Is It between Machine-Generated and Developer-Provided Patches? : An Empirical Study on the Correct Patches Generated by Automated Program Repair Techniques. In *International Symposium on Empirical Software Engineering and Measurement*, volume 2019-Septe. IEEE Computer Society, sep 2019.

[172] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. Automated patch correctness assessment: how far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 968–980, New York, NY, USA, 2021. Association for Computing Machinery.

[173] Suhang Wang, Jiliang Tang, Charu Aggarwal, and Huan Liu. Linked Document Embedding for Classification. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management - CIKM '16*, pages 115–124, New York, New York, USA, 2016. ACM Press.

[174] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 364–374, USA, 2009. IEEE Computer Society.

[175] Weka webite. `https://www.weka.io`, 2024.

[176] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[177] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In Xinyu Wang, David Lo, and Emad Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 479–490. IEEE, 2019.

[178] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 87–98, 2016.

[179] Robert White, Jens Krinke, and Raymond Tan. Establishing multilevel test-to-code traceability links. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 861–872, New York, NY, USA, 2020. Association for Computing Machinery.

[180] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H.D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best Practices for Scientific Computing. *PLoS Biology*, 12(1):e1001745, jan 2014.

[181] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, Franz Wotawa, and Dongcheng Li. Software fault localization: an overview of research, techniques, and tools. *Handbook of Software Fault Localization: Foundations and Advances*, pages 1–117, 2023.

[182] W. Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *Int. J. Softw. Eng. Knowl. Eng.*, 19:573–597, 2009.

[183] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 449–456, 2007.

[184] Huan Xie, Yan Lei, Meng Yan, Yue Yu, Xin Xia, and Xiaoguang Mao. A universal data augmentation approach for fault localization. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 48–60, New York, NY, USA, 2022. Association for Computing Machinery.

[185] Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 226–236, New York, NY, USA, 2017. Association for Computing Machinery.

[186] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–670, 2017.

[187] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings - International Conference on Software Engineering*, pages 789–799. IEEE Computer Society, may 2018.

[188] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 416–426. IEEE Press, 2017.

[189] Xiaofeng Xu, Vidroha Debroy, W. Eric Wong, and Donghui Guo. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering*, 21:803–827, 2011.

[190] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[191] Suresh Yadla, Jane Huffman Hayes, and Alex Dekhtyar. Tracing requirements to defect reports: An application of information retrieval techniques. *Innovations in Systems and Software Engineering*, 1(2):116–124, sep 2005.

[192] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pages 831–841, 2017.

[193] Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. A large-scale empirical review of patch correctness checking approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1203–1215, New York, NY, USA, 2023. Association for Computing Machinery.

[194] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated Classification of Overfitting Patches With Statically Extracted Code Features. *IEEE Transactions on Software Engineering*, 48(8):2920–2938, aug 2022.

[195] He Ye, Matias Martinez, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Softw. Engg.*, 26(2), mar 2021.

[196] Li Yi, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings - International Conference on Software Engineering*, pages 602–614. IEEE Computer Society, jun 2020.

[197] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empir. Softw. Eng.*, 24(1):33–67, 2019.

[198] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering*, 2018.

[199] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, and Xin Xia. Improving fault localization using model-domain synthesized failing test generation. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 199–210. IEEE, 2022.

[200] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Junhao Wen. Improving deep-learning-based fault localization with resampling. *Journal of Software: Evolution and Process*, 33(3):e2312, 2021.

[201] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Xiaohong Zhang. A study of effectiveness of deep learning in locating real faults. *Information and Software Technology*, 131:106486, 2021.

[202] Tony Z. Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. Calibrate Before Use: Improving Few-Shot Performance of Language Models. 2021.

[203] Zhaocheng Zhu and Junfeng Hu. Context Aware Document Embedding. jul 2017.

[204] Yueting Zhuang, Ming Cai, Xuelong Li, Xiangang Luo, Qiang Yang, and Fei Wu. The Next Breakthroughs of Artificial Intelligence: The Interdisciplinary Nature of AI. *Engineering*, 6(3):245–247, mar 2020.