# StandUp4NPR: Standardizing SetUp for Empirically Comparing Neural Program Repair Systems

Wenkang Zhong
State Key Laboratory for Novel Software and Technology, Nanjing University
Nanjing, Jiangsu, China
1007888882@qq.com

Hongliang Ge
State Key Laboratory for Novel Software and Technology, Nanjing University
Nanjing, Jiangsu, China
gehongliang123@163.com

Hongfei Ai
State Key Laboratory for Novel Software and Technology, Nanjing University
Nanjing, Jiangsu, China
ahf876828330@163.com

Chuanyi Li*
State Key Laboratory for Novel Software and Technology, Nanjing University
Nanjing, Jiangsu, China
lcy@nju.edu.cn

Kui Liu
Huawei Software Engineering Application Technology Lab
Hangzhou, Zhejiang, China
brucekuiliu@gmail.com

Jidong Ge*
State Key Laboratory for Novel Software and Technology, Nanjing University
Nanjing, Jiangsu, China
gjd@nju.edu.cn

Bin Luo
State Key Laboratory for Novel Software and Technology, Nanjing University
Nanjing, Jiangsu, China
luobin@nju.edu.cn

## ABSTRACT

Recently, the emerging trend in automatic program repair is to apply deep neural networks to generate fixed code from buggy ones, called NPR (Neural Program Repair). However, the existing NPR systems are trained and evaluated under very different settings (e.g., different training data, inconsistent evaluation data, wide-ranged candidate numbers), which makes it hard to draw fair-enough conclusions when comparing them. Motivated by this, we first build a standard benchmark dataset and an extensive framework tool to mitigate threats for the comparison. The dataset consists of a training set, a validation set and an evaluation set with 144,641, 13,739 and 13,706 bug-fix pairs of Java respectively. The tool supports selecting specific training, validation, and evaluation datasets and automatically conducting the pipeline of training and evaluating NPR models, as well as easily integrating new NPR models by implementing well-defined interfaces. Then, based on the benchmark and tool, we conduct a comprehensive empirical comparison of six SOTA NPR systems w.r.t the *repairability*, *inclination* and *generalizability*. The experimental results reveal deeper characteristics of compared NPR systems and subvert some existing comparative conclusions, which further verify the necessity of unifying the experimental setups in exploring the progresses of NPR systems. Meanwhile, we reveal some common features of NPR systems (e.g., they are good at dealing with code-delete bugs). Finally, we identify some promising research directions derived from our findings.

*Corresponding authors.

## CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

## KEYWORDS

neural program repair, dataset, empirical study

**ACM Reference Format:**
Wenkang Zhong, Hongliang Ge, Hongfei Ai, Chuanyi Li, Kui Liu, Jidong Ge, and Bin Luo. 2022. StandUp4NPR: Standardizing SetUp for Empirically Comparing Neural Program Repair Systems. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556943

## 1 INTRODUCTION

Software bugs costed the global economy 1.1 trillion dollars and affected over 4.4 billion people in 2016, according to the research by Tricentis [51]. Meanwhile, bug-fixing is a time-consuming task which often takes half of a programmer's coding time [3]. So it's no surprise that Automated Program Repair (APR) [43], which aims to repair defective code automatically, has been a hot research topic in the software engineering community. In the past decade, with efforts of researchers in APR field, a bunch of APR techniques have

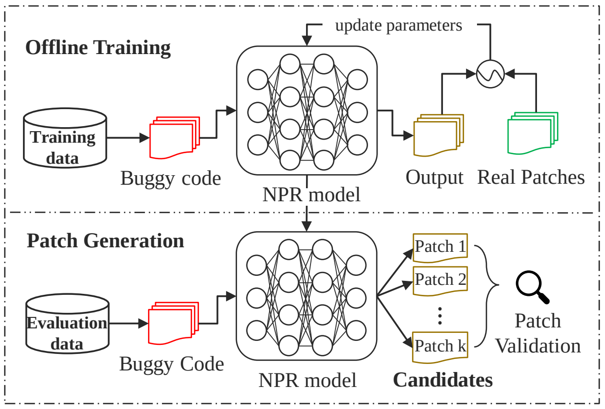


**Figure 1: Training and evaluation process of NPR systems.**

been proposed, which can be categorized into three mainstreams: heuristic-based [15, 39, 41, 48, 52, 57, 65], template-based [21, 22, 24, 31, 32] and constraint-based [10, 40, 59]. Recently, the emerging trend in APR is to apply advanced deep learning techniques in program repair systems, known as Neural Program Repair (NPR) [2, 4, 5, 7, 8, 18, 27, 36, 42, 50, 54, 55, 67]. The NPR approaches frame the bug-fixing process as a translation task from defective code to correct code, employing the neural machine translation models [1] that are popular in the field of Natural Language Processing (NLP). Compared with the previous APR approaches, a huge advantage of NPR systems is their low dependence on domain knowledge and extra resources such as the test suites. Consequently, more and more researchers are paying attention to this field and a bench of novel NPR systems have been proposed.

Though successful, a question of great concern remains to be answered: **how far has the NPR field progressed now?** To answer this question, it is unavoidable to compare different NPR systems from multiple perspectives such as repairability, inclination and generalizability. However, at the moment, there is a big challenge in making such a comparison: previous NPR systems are trained and evaluated in very different setups. Table 1 provides a summary of detailed setup information for previous NPR systems. As we observed, such differences can be summarized as three points: (1) **very-different training data**, (2) **inconsistent evaluation data** and (3) **wide-ranged candidate numbers**. For example, CoCoNut [36] is trained on 3,241,966 samples and evaluated on 393 bugs of Defects4J[20] and 40 bugs of QuixBugs [29], while SequenceR [5] is trained on 35,578 samples and evaluated on 75 bugs of Defects4J [20]. For each bug, CoCoNut [36] generates 1,000 candidates for evaluation while SequenceR [5] only generates 50 candidates. We provide a detailed discussion on the above three differences in Section 2.

Such differences may bring huge threats when comparing NPR systems to draw some domain-level conclusions. Next, we explain this following the training and evaluating process presented in Figure 1: (1) First, for learning based methods, a well-known fact is that the content and size of training data have a great impact on their performance [14, 25, 26]. As a kind of learning-based approach, each NPR model has a offline learning phase. During this phase, the NPR model depends on samples from training data to optimize its parameters. In a word, training data determines the absolute performance of a NPR model. (2) To measure the performance, NPR systems are evaluated on specific dataset to report a quantified result of their repairability. Obviously, different evaluation sets will result in different measurements. (3) During the prediction phase, the NPR system generates patches with the top confidence score, forming top-k candidate patches for each buggy input. Such candidates are then validated by executing a test suite and further perform a manual check. If one of the k candidates passes the validation, the model is considered to have successfully fixed the bug. Since NPR systems are a kind of probability model, a larger candidate set surely has a higher probability to fix the bug. **Thus, when comparing NPR systems that have the above three differences on settings, we can not draw fair-enough conclusions. The above three factors must be set to the same value for an in-depth analysis of the existing NPR systems.**

In this paper, we perform an empirical comparison among the six SOTA NPR systems[4, 5, 8, 36, 54, 67]. To mitigate the potential threats which the above three factors may bring on validity of the comparison, we build a new benchmark dataset for NPR firstly, named *NPR4J-Benchmark*. The benchmark contains 144,641 standard data samples of Java for training, 13,739 for validation and 13,706 for evaluation. We focus on Java bugs for one-line type since it is the most popular scene researched in previous APR studies. To ease the future research, we also implement an extensive framework that supports training, reusing and evaluating NPR systems, named *NPR4J-Framework*.

Then, based on the benchmark and framework, we conduct a huge experiment to explore the current state of the six NPR systems. Inspired in previous work that empirically evaluates test-based APR systems [9], we design specific experiments to investigate the *repairability*, *inclination* and *generalizability* of NPR systems. Concretely, we aim to provide the answers to the following research questions:

- **[RQ1] Repairability**
  (1) *How many bugs in NPR4J-Benchmark can be fixed by the six NPR systems?*
  (2) *How does the candidate number influence the repairability of the NPR systems?*
- **[RQ2] Inclination**
  (1) *When feeding the same training data, will NPR systems tend to fix the same bugs?*
  (2) *Do NPR systems have a repair preference for bug types?*
- **[RQ3] Generalizability**
  *RQ3 Can the NPR systems fix the bugs which have never been seen during training?*

With respect to repairability, we first observe that the best NPR system can repair 22% bugs of *NPR4J-Benchmark*. By comparing with original evaluations, we find that different setups can lead to different conclusions among the NPR systems. For example, the performance ranking between models changes when the candidate number is set from 1 to 100. This finding verify the necessity to standardize setups for comparing NPR systems. Then, regarding inclination, we find that NPR systems share a 7%-39% unique patching rate, indicating that combing the advantages of various

**Table 1: Training and evaluation setups of nine NPR systems for Java.**

| NPR System | Training Source | # Training Instance | Evaluation Dataset | # Evaluation Instance | Candidate Number |
|---|---|---|---|---|---|
| CoCoNut [36] | commits before 2010 on GitHub, projects from GitLab and Bitbucket | 3,241,966 | Defects4J [20]<br>QuixBugs [29] | 393<br>40 | 20,000 |
| CODIT [4] | 6 projects from Defects4J [20] | 22,060 | Defects4J [20] | 117 | 5 |
| Cure [18] | CoCoNut's training data | 4,040,000 (pretrain)<br>2,720,000 (finetune) | Defects4J (V1.4)<br>QuixBugs [29] | 393<br>40 | 5,000 |
| DLFix [27] | BigFix: building from a bug detection dataset [28] | 20,000 | Defects4J [20]<br>Bugs.jar [47]<br>BigFix[27] | 101<br>1,158<br>2,176 | 10 |
| Edits [8] | 10,235 most-starred Java repositories on GitHub | 55,000 | EditsDataset[8] | 5,000 | 25 |
| Recoder [67] | Java projects between 2011 and 2018 on GitHub | 82,868 | Defects4J (v1.4)<br>Defects4J (v2.0)<br>QuixBugs [29]<br>IntroClassJava [11] | 395<br>420<br>40<br>297 | 100 |
| Tufano [54] | BFP: commits between 2010 and 2017 on GitHub | 46,680 (small)<br>52,364 (mid) | BFP (small)[54]<br>BFP (mid)[54]<br>CodRep (small)[6]<br>CodRep (mid)[6] | 5,835<br>6,545<br>3,027<br>6,205 | 50 |
| SequenceR [5] | original source of BFP [54] | 35,578 | Defects4J[20]<br>CodRep [6] | 75<br>4,711 | 50 |
| Tang [50] | small version of BFP [54] | 46,680 | BFP (small) [54] | 5,835 | 5 |

systems may be a viable approach for improving performance. In addition, we find that the current NPR systems are good at generating code-removal patches but are poor at fixing complex bugs that require multiple types of editing operations (e.g., deleting a token followed by inserting two tokens). In terms of generalizability, we find that NPR systems are capable of fixing bugs that they have not encountered during training. Moreover, the NPR systems are more effective on bugs that have a similar sample in the training data, which suggests that collecting more data for training could be a practical measure to improve their performance.

In summary, we make the following contributions:

(1) A well-organized benchmark, named *NPR4J-Benchmark*. The benchmark provides standard bug-fix samples of Java for training, validating and evaluating NPR systems (144,641 for training, 13,739 for validation and 13,709 for evaluation). Samples are divided into the three subsets following certain criteria to avoid potential data issues such as data leaking.
(2) A framework tool, named *NPR4J-Framework*. The framework wraps six NPR systems' original codes into unified interfaces, supporting training and using trained models to fix bugs in an easy-to-use way. Meanwhile, it is extensive to add new NPR systems.
(3) A novel analysis in NPR field, which provides some interesting findings on *repairability*, *inclination* and *generalizability* of NPR systems when comparing them under same setups, as well as several promising venues derived from the findings for future research.

All codes and data of *NPR4J-Benchmark* and *NPR4J-Framework* are publicly available[1] for the future research.

The remainder of this paper is organized as follows. Section 2 illustrates our observations on setups of the previous neural program repair systems, which points out the threats that should be excluded in the experiment. Section 3 explains our detailed study plan to support our empirical experiments. Section 4 presents the experiment results and points out the possible research venues deriving from our findings. Section 5 discusses threats to validity. Finally, we introduce related works in Section 6 and summarize our work in Section 7.

## 2 DIFFERENCES ON SETUPS OF NPR SYSTEMS

Table 1 summarizes our review on training and evaluation setups of nine NPR systems. To gather these NPR researches, we first search the living review of Automated Program Repair [43] and a previous summary of NPR systems [66] to get a list of NPR studies [2, 4, 5, 7, 8, 18, 27, 36, 42, 50, 54, 55, 67]. We focus on NPR systems for Java because Java is the most popular language researched in previous APR studies. Thus, the NPR systems that are not evaluated on Java are discarded [2, 7, 55]. Each row in Table 1 provides the five categories of information: (1) the source of training data, (2) the number of instances for training, (3) the source of evaluation data (4) the number of instances for evaluating and (5) the setting of the candidate number. Through the table, we find following three main differences on **training data**, **evaluation data** and **candidate numbers**:

*Difference #1: NPR systems are using different training data collected from different sources.* As shown in Table 1, four of the nine NPR systems [8, 36, 54, 67] mine bug-fix pairs from code repositories and build their own datasets. Such datasets are also used by subsequent NPR systems [4, 5, 18, 50]. Although most researchers choose GitHub[2] as their data source, the concrete code projects used in each dataset are clustered by different criteria. To summary, they are either time-divided (before 2010 [18, 36], 2010-2017

[1]https://github.com/kwz219/NPR4J

[2]https://github.com/

[50, 54], 2011-2018 [67]), star-ranked [8] or derived from existing datasets [4, 5, 27]. Even though the data sources are the same, the final amount of training instances each system uses can be variant (e.g., Cure [18] and CoCoNut [36]).

*Difference #2: Evaluation datasets are sparsely used and the instances that are used by each NPR system for evaluation may differ even they belong to the same dataset.* In total, we found the eight different datasets (Defects4J [20], QuixBugs [29], Bugs.jar [47], BigFix [27], EditsDataset [8], IntroClassJava [11], CodRep [6], BFP [54]) being used in the evaluation process. The first issue is that NPR systems are evaluated sparsely on those datasets. In terms of frequency, Defects4J [20] is the most commonly used dataset that has been used by six NPR systems. The other seven datasets are used only 1-3 times. It means currently, if we want to compare the existing NPR systems, their reported performance on Defects4J[20] are the only evidence we have. However, as shown in a previous study [9], using a single benchmark when evaluating repair tools, a bias can be introduced, which makes it hard to generalize the performance of repair tools. Secondly, some of the NPR approaches use different parts of the same dataset to evaluate their models. For example, CoCoNut [36], CODIT [4], DLFix [27], Recoder [67] and SequenceR [5] all get evaluated on Defects4J [20], but the instances used for evaluation are different, ranging from 75 samples to 395 samples. On the one hand, the choice of samples for evaluation may lead to a bias. In addition, for the NPR studies, a common approach to compare with others is to copy their reported results directly, ignoring the fact that they may have used a different part of the benchmark. Thus, their measured performance may not be consistent when comparing on the same base. The third issue is that not all NPR studies illustrate how they exclude evaluation-related bugs from the training data. According to previous study[35], 70% of the code on GitHub contain clones of previously created files, which may lead to a data leakage problem if the NPR studies do not follow strict strategies to clean their training data.

*Difference #3: When predicting, the number of candidate patches generated by the NPR systems varies widely.* In the evaluation process, for each bug, the NPR systems generate 5 to 5,000 candidates for validation. Since the NPR systems are a kind of probabilistic generative model, a large candidate set obviously has a higher probability to contain the correct patch. Therefore, the NPR system can achieve higher performance with a larger amount of candidates. However, there is a ignored price at the time cost of model executing and patch validation. Considering that in a real-world scenario, each candidate need to be executed on test cases or checked manually, this time price can be seen as a linear growth with the increase of candidate number. Another factor that makes the comparison on NPR systems difficult is that some studies [18, 36, 67] didn't report their performances on a smaller candidate size setting.

**Conclusion:** The above three differences on setups indicate that the previous NPR systems are not compared on the same scale. Potential threats on validity may be brought by different settings on *training data*, *evaluation data* and *candidate number*.

**Our Solution:** In this paper, we aim to provide a domain-level view of the NPR field through an empirical comparison of the existing NPR systems. To mitigate the threats aforementioned, we build a new benchmark for the NPR that contains uniformed training and evaluation data. Besides, we develop a framework tool that defines a pipeline for training and evaluating the NPR systems. Based on the benchmark and the framework tool, we perform a large experiment that evaluates NPR systems at the same scale.

## 3 STUDY DESIGN

This section illustrates our detailed study designed for the empirical comparison of the NPR systems. Section 3.1 lists the research questions used in analyzing each NPR model from three perspectives. Section 3.2 briefly introduces the selected six NPR systems. Next, we explain how we build the two milestones that support our experiments - dataset (in section 3.3) and framework (in section 3.4). Finally, section 3.5 introduces our experiment setup.

### 3.1 Research Questions

- **RQ1 Repairability**
  (1) *How many bugs in NPR4J-Benchmark can be fixed by the six NPR systems?* The purpose of this question is to measure the repairability of six NPR systems on a diversity of bugs. Additionally, we intend to verify the necessity for standardizing setups by comparing the results from our experiment with those from the original experiment.
  (2) *How does the candidate number influence the repairability of the NPR systems?* We design this question based on the observation that the previous NPR systems use wide-ranging candidate numbers, which enables us to investigate the impact of candidate numbers on repairability and see whether they affect comparison results across the NPR systems.
- **RQ2 Inclination**
  (1) *When feeding same training data, will NPR systems tend to fix the same bugs?* We seek to answer this question by calculating the overlapping and unique patching rates of each NPR system. This can be a reference for other researchers to improve the performance of NPR. For example, if the unique rate is high, combining the benefits of various NPR systems could be advantageous.
  (2) *Do NPR systems have a repair preference for bug types?* This question examines whether the NPR system has a comparative advantage in fixing certain types of bugs.
- **RQ3 Generalizability**
  *Can the NPR systems fix the bugs which have never been seen during training?* This question is to investigate whether the NPR systems have the ability to repair the unknown bugs.

### 3.2 Subject NPR Systems

To include NPR systems in our empirical study, we first search the living review of APR [43] and get a list of NPR studies [2, 4, 5, 7, 8, 16, 16, 18, 27, 36, 42, 50, 54, 55, 60, 60, 61, 61, 67]. Since we focus on the NPR systems repairing dynamic defects in Java programs, studies focusing on fixing other-type errors [16, 60, 61] and other programming languages [2, 7, 55] are excluded during the first round of filtering. After that, we end up with nine NPR systems, as shown in Table 1. Next, We propose the two criteria for selecting candidates among the nine NPR systems to further ensure the success of this empirical comparison:

- *Availability*: The source code of the NPR system should be available. Thus we exclude Cure [18] and Tang et al. [50]'s model.

- *Executability*: Some APR approaches provide publicly available source codes, which however cannot be executed. We have tried our best to fix inexecutable approaches. But unfortunately, we still fail to make it a success for DLFix [27], so we exclude it.

Eventually, we derive a baseline package containing the six NPR systems. Next, we briefly describe each selected NPR system.

**Tufano** [54] trains a RNN-based Encoder-Decoder model which is able to translate the entire buggy method into a fixed version. To reduce the difficulty of learning, they abstract identifiers and literals in the buggy code to simplify the input and output. During the abstraction process, the source code is firstly fed to a Java parser, which recognizes the identifiers and literals in the stream. Then the parser generates and substitutes a unique ID for each identifier/literal within the buggy context. Only the most frequent words are kept. Such abstraction reduces the model's choices when generating patches, therefore increasing the patching rate.

**SequenceR** [5] fixes bugs based on sequence-to-sequence learning on source code. Compared with Tufano's model [54], it uses a additional copy mechanism [49] to overcome the unlimited vocabulary problem that occurs in handling big code. The model takes the abstract context of the buggy line as the input and predicts the fixed result of that line. The abstract buggy context consists of line-, method-, and class-level information of the buggy line. Specifically, at line level, special tokens are inserted before and after the buggy line to indicate the location of the bug. Then, the remainder of the buggy method is kept in the representation. Finally, all the instance variables and initializers, along with the signature of the constructor and non-buggy methods from the buggy class are added to the input unless reaching a truncation limit.

**CoCoNut** [36] uses ensemble learning on the combination of Convolutional Neural Networks (CNNs) [12] and a context-aware neural machine translation architecture to automatically fix bugs for multiple programming languages. To better represent the context of a bug, it introduces a new context-aware architecture that represents the buggy source code and its surrounding context separately. Such architecture enables the model to distinguish the buggy line and the context better. To reduce the size of vocabulary, it leverages the world-level tokenization by considering underscores, camel letters, and numbers as separators.

**CODIT** [4] decomposes the task of fixing the buggy line into two stages and each stage uses one LSTM-based Neural Machine Translation Model. The first stage is to generate the sequence of grammar rules for constructing the CFG (i.e., Context Free Grammars) of the fixed code line. Upon the rules are generated, the CFG can be constructed to retrieve the type of code tokens occurring consecutively in the fixed line. The next stage is to predict the ultimate fixed code tokens according to the code tokens and corresponding token types of the buggy line, and the types of code tokens predicted in the first stage.

**Edits** [8] models the patch generation process as performing token-level insertion and deletion operations on the buggy code. Edits adds the two additional pointers[49] to Transformer[56], implementing the editing operation of "insertion" and "deletion". Edits takes the buggy code line as the input. When generating patches, it outputs edits operations on the buggy code, each indicating the insertion start position, the removal end position and any missing tokens.

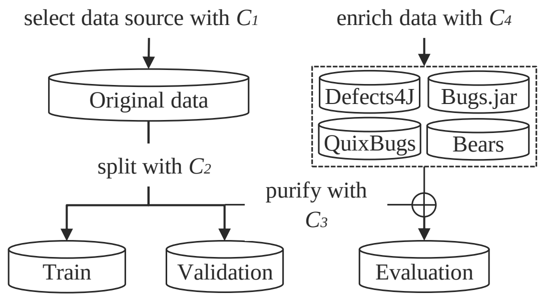


**Figure 2: Construction process of our benchmark data.**

**Recoder** [67] designs a syntax-guided decoder to generate edits on the AST of the buggy method. It takes (1) AST traversal sequence, (2) Tag Embedding and (3) AST-based Graph as inputs. During inference, Recoder use a novel provider/decider architecture to output edit operations on the current AST. It uses three providers to represent insertion, modifying and copying operations. Providers are responsible to provide a set of choices for expanding a non-terminal of the current AST. The decider estimates the likelihood of using each provider.

Among the six NPR systems include in our experiments, five of them ([4, 5, 8, 36, 67]) can only deal with one-line bugs (in a buggy method, only one code line is buggy). Tufano [54] is the only one that can repair any bugs within one method. It can be used to fix one-line bugs without any modification of the original model.

## 3.3 Dataset Construction

To run our empirical experiments, we construct a new benchmark dataset instead of reusing any of the existing datasets. The reasons are two-fold. First, to avoid data leakage, samples related to bugs in the evaluation set should be excluded from the training dataset. However, existing training data that the previous NPR systems use may not provide enough meta information to do the exclusion. Second, some existing datasets may pose potential threats to experimental performance due to some issues on data. For example, although CoCoNut [36] reports that the training set they use (Java 2006) contains over three million samples, we find that nearly one-third of the samples are duplicated pairs and non-character changes, through checking their raw data by string matching. We argue that for mitigating threats brought by data issues, the experiment training and evaluation data should satisfy the following criteria:

- *Criterion #1: Each data sample from the data source should provide enough meta information.*This criterion is designed for selecting a data source. The data source we used in this experiment must provide enough meta information to ensure the data traceability. For NPR, such traceability is important. For example, meta information such as the repository of bugs should be provided to to exclude bugs from the training data that belong to the same repository as bugs for evaluation.

- *Criterion #2: Evaluation-related Bugs should not appear in the training set.* Considering that codes within the same projects may contain some cheating information which is inaccessible for the NPR systems under empirical scenarios, we perform a strict strategy, dividing data by projects.
- *Criterion #3: Each bug in the evaluation set should be attached to a corresponding issue or bug report.* This criterion is to ensure that the bugs we use for evaluation are highly reliable samples from the real world.
- *Criterion #4: The benchmark should be peer-reviewed and contains human-written patches for each bug.* For perfect fault-localization, we need human-written patches to locate the buggy position of the source program.

With the guidence of above four criteira, we construct a new benchmark dataset called *NPR4J-Benchmark*, following the process shown in Figure 2. The overall process can be summarized into the following three steps:

**Step 1: Selecting the data source.** It is to collect enough bug-fix pairs. A common way is to crawl large amounts of data from the code repositories such as GitHub. However, it can be laborious and time consuming. Fortunately, previous studies have done this work [8, 36, 54, 67]. To select proper data sources from them, we obey $C_1$. Among the four data sources, we select source of BFP [54] as our original data source, which contains 787,178 bug-fixing commits for java. Each data sample is attached with meta information including repository, commit message and commit url.

**Step 2: Splitting sub-datasets.** This is to construct three sub-datasets: *training*, *validation* and *evaluation*. The training set contains the samples that the NPR systems rely on to learn to fix bugs. During the offline training process, the NPR systems can get a early evaluation on the validation set to modify hyper-parameters of repair models. Eventually, the evaluation set is used to measure the performance of trained the NPR systems. One well-known basic rule is that there should be no overlap between data in the training, validation and evaluation sets. We obey $C_2$ for the splitting phase. Concretely, we follow the practice by Recoder [67], excluding data samples which belongs to a clone project of projects that evaluation data uses or a program repair project that use these projects from the training set.

**Step 3: Purifying and enriching evaluation resources.** To construct a more-diverse benchmark for evaluation, we collect bugs from multiple sources. The first source is the data source which has been selected in the first step. According to previous research [19], bug-fix commits without peer-reviewing may contain bug-irrelevant changes and some of the commits are of low quality. Thus, to ensure the reliability and quality of bugs in the evaluation set, we follow $C_3$ to purify bugs. First, We perform a regular-match on the commit message of each bug-fix commit. Then, we only keep data samples that identify explicit issues or bug ids in the commit message. Next, we select the existing benchmarks in APR as additional data sources, following $C_4$. With the guidance of $C_4$, Defects4j [20], Bugs.jar [47], QuixBugs [29] and Bears [37] are added to our evaluation set.

The final statistics for the benchmark dataset are listed in Table 2. In total, we collect 144,641 bug-fix samples in the training set and 13,739 in the validation set. Each sample consists of six types of information: (1) the buggy line, (2) the fix line, (3) method-level context, (4) class-level context, (5) fault location and (6) meta information. For evaluation, we collect 13,706 bugs in total. Specifically, our evaluation set consists of two parts: Diversity and Empirical. Diversity provides 13,295 bugs (12,815 bugs from BFP [54] and 480 bugs from [47]). For empirical validation, we collect 411 bugs (119 bugs from Bears [37], 260 bugs from Defects4J [20] and 32 bugs from QuixBugs [29]) with corresponding test suites in the empirical part. In the evaluation set, the 12,815 bugs from BFP [54] are one-line bugs that can be fixed by a single line modifying within one method. To enrich the number of bugs provided by established benchmarks, the other part are bugs that can be fixed by single-line modifying on one or multiple methods (a bug may consist of single-line errors in multiple methods.).

**Table 2: Statistics of *NPR4J-Benchmark*.**

| Source | Training | Validation | Evaluation | |
|---|---|---|---|---|
| | | | Diversity | Empirical |
| BFP [54] | 144,641 | 13,739 | 12,815 | — |
| Bears [37] | — | — | — | 119 |
| Bugs.jar [47] | — | — | 480 | — |
| Defects4J [20] | — | — | — | 260 |
| QuixBugs [29] | — | — | — | 32 |
| Total | 144,641 | 13,739 | 13,295 | 411 |

## 3.4 Framework Implementation

Although most NPR systems open source their codes, we notice that there exists some usability issues proposed in their GitHub repositories (For example, there are open issues concerning the evaluating procedure, data availability, and hyper-parameter settings, etc., of CoCoNut[3]). One major reason is that codes of the NPR system are often very complex. For example, their codes are often made up of multiple programming languages (i.e., java for code preprocessing and python for deep learning). The python part is implemented with various deep learning frameworks (PyTorch[4], Tensorflow[5], OpenNMT[6], Fairseq[7], etc.). Thus, modifying their codes to satisfy new requirements (i.e., training on a new dataset) requires users' expertise on both program repair and deep learning fields. To ease our experiments and benefit future researches on NPR field, we wrap original codes of six NPR systems (Tufano [54], SequenceR [5], CoCoNut [36], Codit [4], Edits [8], Recoder [67]) into a new framework, called *NPR4j-Framework*. The overall workflow of *NPR4j-Framework* is shown in Figure 3. *NPR4j-Framework* supports training NPR systems from scratch and using trained systems to fix bugs empirically. All codes are organized into three main components: DataManager, Preprocessor and Trainer. DataManager is responsible for processing the original data into a standard data form which we named, diff. The core part of the DataManager is the DiffParser. The DiffParser parses each pair of buggy and fixed Java class files into diffs, each of which represents the exact line-level difference between the

[3]https://github.com/lin-tan/CoCoNut-Artifact/issues
[4]https://pytorch.org/
[5]https://www.tensorflow.org/
[6]https://opennmt.net/
[7]https://github.com/pytorch/fairseq

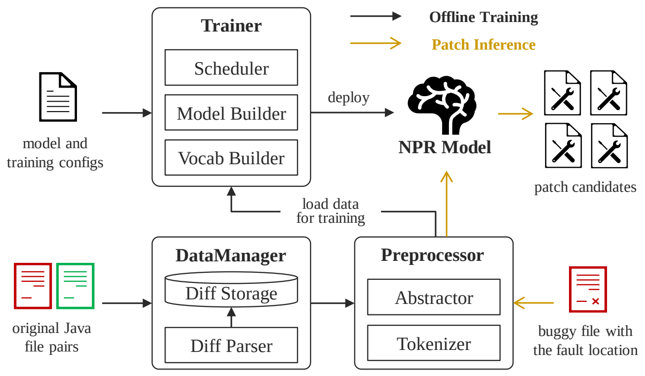


**Figure 3: The workflow of *NPR4J-Framework***

buggy and the fixed program, and is associated with the position of the buggy line. Preprocessor implements a variety of processing methods that are designed by the six NPR systems to represent the buggy code. It consists of a Tokenizer and an Abstractor. Tokenizer is responsible for dividing textual codes into a list of tokens. And the Abstractor supports common code abstraction operations that most NPR systems use to simply the codes, such as replacing the string elements with constant ids. Once the preparation of data ends, Trainer is pressed into service to train the NPR model. During this phase, all parameters such as the hyperparameters of the model should be written on a configuration file. Then, the configuration file will be delivered to the Vocab Builder and the Model Builder to configure the vocab and initialize the model for training. The Scheduler determines when to stop the training.

Besides training and evaluating a NPR system with the data provided in *NPR4J-Benchmark*, the framework also allows users to train the NPR systems on their own datasets. Users just need to provide pairs of buggy-fix java class files and use the DataManager and the DiffParser to process the data. What's more, our framework is extensive. The well-defined interface makes it easy to add a new NPR system.

## 3.5 Experimental Setup

**Experiment Environment:** We run all our experiments on four machines, including training and evaluating all NPR systems. Each machine has four GPUs (NVIDIA Tesla V100 SXM2 32GB) and a 20-core CPU (Intel(R) Xeon(R) Gold 6248 CPU).
**Fault Localization:** All bugs we use in this experiment are perfectly fault-localized. We identify the buggy line for every bug with the help of human-written patches provided in *NPR4J-Benchmark* and the DiffParser component in *NPR4j-Framework*.
**Training:** To obtain well-trained models, we use random search strategy to tune hyperparameters of NPR models. For each NPR model, we first train a model with original hyperparameters they reported in the paper (if provided). Then, we perform a random search on adjustable parameters of each model (embedding size, vocab size, learning rate). For each NPR approach, we select one which performs best on the validation set as the final model to get evaluation. Specifically, since CoCoNut [36] is an ensemble model, we set the ensemble size to 10, which is the same as its original setting.
**Inference:** In inference mode, we set the beam size to 100 and set the candidate number to 100, which means for every bug in the evaluation set, each NPR system will generate 100 candidate patches for validation. Especially, CoCoNut [36] ensembles 10 models in the original paper. Therefore, we also generate 100 patches for each single model of CoCoNut according to the ensemble setting.
**Patch Validation:** After generation, all candidates need to be validated by executing on test cases. We stop validation when obtaining the first plausible (test-adequate) patch.
**Patch Assessment:** Typically, APR tools are evaluated empirically. If a patch pass all test cases, it is regarded as *plausible*. Such *plausible* patches require manual assessment to make sure that they are *correct*, considering potential faults they may bring. However, manual assessment can introduce biases, as mentioned in previous study [63]. Another way to assess the correctness of one patch is to check if it is identical to the human-written patch. This way is more objective but may loss some correct patches that have equal semantics with human-written patches. In our experiment, to reduce the bias of assessment, we use both two ways to assess the correctness of NPR systems' predictions. First, for bugs in the diversity part of *NPR4J-Benchmark*, a patch is regarded as correct if it's identical to the human-written patches. Second, for bugs in the empirical part, a patch is regarded as correct only if (1) it's identical to the human-written patches or (2) it can pass the corresponding test cases and is checked by at least two of the authors.

## 4 STUDY RESULTS AND DISCUSSION

### 4.1 Repairability

*RQ1.1 how many bugs in NPR4J-Benchmark can be fixed by the six NPR systems?*

*Method and Results.* We empirically retrain and evaluate the six NPR systems on *NPR4J-Benchmark*. Table 3 reports the number of patches correctly predicted by the six NPR systems, with a simple comparison to their original evaluation results. For the diversity part, we count systems' perfect predictions that are identical to the human-written patches. For the empirical part, we count the number of correct/plausible patches. For comparison, we recount their original results on the same base ( bugs used for evaluation in *NPR4J-Benchmark*). For example, Recoder [67] fixes the bug "Chart-3" of Defects4J [20] in their original experiment, however "Chart-3" is not in our 260 bugs used in Defects4J [20]. Such fix will not be counted. This situation is caused by the fault-localization method we use in this experiment. We aim to get precise line-level fault position through comparing the human-written patch with buggy program. In some cases, a bug can be fixed in just modifying one line but its human-written patch contains multi-line changes. We can not identify such bugs when collecting data for *NPR4J-Benchmark*.

*Finding 1.* With unified setups on training data, evaluation data and the candidate number, we obtain a result that differs from the original evaluation of the six NPR systems, emphasizing the necessity to standardize experimental setups when comparing NPR systems. In our experiment, SequenceR [5] is the best system that repair 22% out of 13,706 bugs in *NPR4J-Benchmark*. CoCoNut [36] and

**Table 3: Repairability of six NPR systems measured in our experiment and their original evaluations. For the diversity part, we count patches that are identical to human-written patches. For the empirical part, we count the number of correct/plausible patches in our experiment and recount the number of correct patches in their original experiment.**

| | | Diversity | | Empirical | | |
|---|---|---|---|---|---|---|
| | NPR System | Main | Bugs.jar | Defects4J | Bears | QuixBugs |
| | | 12,815 bugs | 480 bugs | 260 bugs | 119 bugs | 32 bugs |
| original | CODIT | – | – | 10 | – | – |
| | Recoder | – | – | 47 | – | <17 |
| | CoCoNut | – | – | 34 | – | 12 |
| | SequenceR | – | – | 13 | – | – |
| in our experiment | CODIT | 96 | 8 | 5/6 | 1/1 | 1/1 |
| | Edits | 179 | 20 | 15/20 | 2/7 | 6/6 |
| | Tufano | 1,080 | 43 | 25/31 | 9/18 | 7/8 |
| | Recoder | 1,538 | 34 | **46/57** | 5/17 | 10/11 |
| | CoCoNut | 2,403 | 57 | **48/60** | **19/33** | 13/13 |
| | SequenceR | **2,900** | **56** | **48/57** | 16/26 | **15/16** |
| | Total | 4,526 | 94 | 73/81 | 23/37 | 22/22 |

Recoder [67] are two systems that show comparable performances on Defects4J [20] or Bears [37]. However, in original evaluations, we find Recoder [67] is the best system that repair 47 bugs on the 260 bugs, outperforming CoCoNut [36] (34 bugs) and [5] (13 bugs). In a concrete case of Recoder [67], though the candidate number set by their original paper is same as ours (100), but we also observe a large difference on the performance. On QuixBugs [29], Recoder [67] can only fix 10 bugs in our experiment, while in its original results the number is less than 17 (we can not get a precise number since the authors haven't release concrete patches on QuixBugs [29] until this paper is written). On Defects4J [20], though the number of correct fixes has little change (47 to 46), we find that only 61% bugs are overlapped. This finding indicates that the model can produce big differences in performance depending on the setup, further result in a different comparison result. In theory, considering that these NPR systems are based on deep learning technologies, when the training data and evaluation data change, the model performance will surely change [25, 26]. However, from a practical perspective, it's important to figure out what cause such changes because they can make NPR systems not robust when fixing different bugs. Imaging that when a NPR system is used in a real-world scenario, the users will never want the system to be unstable and unpredictable. Thus, we put forward a question on NPR field which requires further exploration: does each NPR system have a particular preference for training data or for the types of bugs repaired, resulting in a different learning result based on different training data ? We believe the answer to this question could provide some practical tricks to improve NPR systems' performance.

*RQ1.2 How does the candidate number influence the repairability of the NPR systems?*

*Method and Results.* We evaluate NPR systems' fix rates under different candidate numbers on the main part of diversity (12,815 bugs). Figure 4 shows the fix rates of six NPR systems under different candidate numbers. We set the candidate numbers from 1 to 100 with an interval of 5. The fix rate represents the ratio of perfect predictions. Fixing one bug under candidate size $K$ means within the $K$ candidates, at least one is identical to the human-written patch.

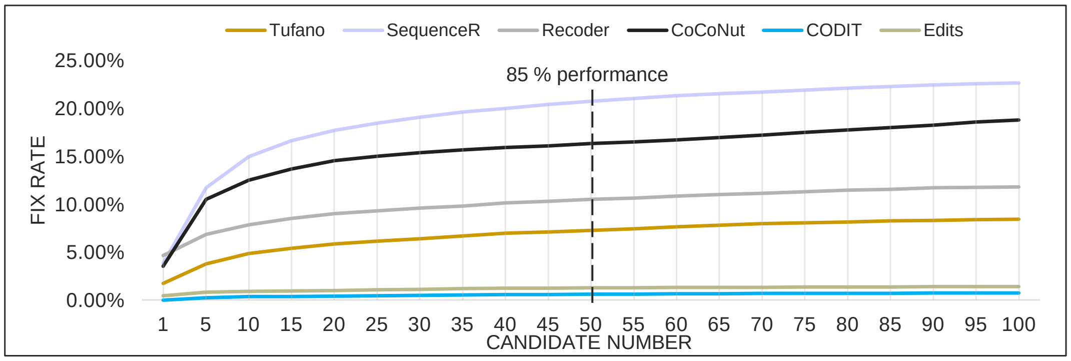
*Finding 2.* Under different candidate numbers, the comparison results of NPR systems can be different, even when they are trained and evaluated with unified dataset. As shown in Figure 4, when the candidate number is limited to 1, Recoder [67] performs best out of the six systems. As the candidate number growing, the performance gain of Recoder [67] is lower than that of SequenceR [5] and CoCoNut [36]. When the candidate number reaches 100, SequenceR [5] is the one that behaves best. We notice that several previous NPR systems [18, 36, 67] don't report their repairability on a low candidate number that are comparative with others. Thus, it is difficult to say which NPR System is better if they use different settings for the candidate size, since we can not know whether the performance difference comes from a different candidate size or from the other components of the model. This finding verify the necessity to compare various NPR systems under same candidates numbers. Furthermore, we suggest that for making a detailed comparison for NPR, future studies should report their performances on different candidate numbers.

*Finding 3.* The performance of the NPR system does not improve linearly with the increase in the candidate number. Through Figure 4, we find that the performance of the NPR system is significantly improved when candidate number is 100 compared to that when candidate number is 1. However, the performance does not increase linearly. To make this point intuitive, we draw a performance separation line in the Figure 4. We find that, when the candidate number is set to 50, all NPR systems have already reached 85% (+-5%) performances of that under a candidate number of 100. Considering that under a real-world bug-fixing scenario, each candidate needs to be executed on test cases and checked manually. The time cost can be seen as a linear growth with the increase of the candidate number. Inspired by this finding, here we throw an empirical question for future research: how to balance the time cost and the fix rate? For example, for bugs in Bears [37] , it usually takes one to a few minutes to compile and run the test cases of the buggy project. It seems affordable in our experiment since the max candidate number is 100. However, some NPR systems have use a much more larger candidate number (1,000 in CoCoNut [36] and 5,000 in Cure [18]). Such huge candidate numbers can surely bring a better performance, but meanwhile means that in a worst situation (the correct patch locates in the latter part of candidates), validating candidates for one bug may cost several days. Is it worthwhile to fix additional little bugs at a much higher cost in time? We suggest follow-up researchers should pay attention to this question.

## 4.2 Inclination

*RQ2.1 When feeding the same training data, will NPR systems tend to fix the same bugs?*

*Method and Results.* We calculate the overlapped and unique patching rate of six NPR systems, as shown in Table 4. Each row presents the percentage of overlapped patched bugs of one NPR system with the rest of the systems. The diagonal of the table represents the rate of fixes that can only generated by the tool.



Ranks by fix@1:
-----------------------------
Recoder: 4.64%
SequenceR: 3.82%
CoCoNut: 3.51%
Tufano: 1.72%
Edits: 0.46%
CODIT: 0.01%

Ranks by fix@100:
-----------------------------
SequenceR: 22.63%
CoCoNut: 18.75%
Recoder: 11.77%
Tufano: 8.43%
Edits: 1.40%
CODIT: 0.75%

**Figure 4: Influence of different candidate numbers on the repairability of the six NPR systems. The candidate number is ranged from 1 to 100.**

**Table 4: Overlapping and unique patching rate of six NPR systems.**

| | CODIT | Edits | Tufano | Recoder | CoCoNut | SequenceR |
|---|---|---|---|---|---|---|
| CODIT | 7% | 5% | 61% | 22% | 69% | 85% |
| Edits | 2% | 9% | 35% | 24% | 67% | 78% |
| Tufano | 5% | 5% | 17% | 22% | 61% | 75% |
| Recoder | 1% | 2% | 15% | 39% | 31% | 54% |
| CoCoNut | 2% | 5% | 27% | 19% | 27% | 66% |
| SequenceR | 2% | 4% | 28% | 29% | 55% | 24% |

For example, 69% of bugs patched by CODIT (row 2) can also be patched by CoCoNut (column 5). Among the bugs fixed by CODIT, 7% (row 2, column 2) can't be fixed by any other NPR systems.

*Finding 4.* First, we note that there is indeed an overlap between bugs fixed by each NPR system. Regarding the fix results we obtain in Table 3, we find that overlapped rate is correlated to the repairability of NPR systems. Generally, the more bugs one NPR system can fix, the higher rate other systems overlap with it. For example, SequenceR [5] and CoCoNut [36] reach the best and second-best performance on the main part bugs. Relatively, the overlapping rate of other systems with SequenceR [5] and CoCoNut [36] are much more higher. This is reasonable since NPR systems have many commonalities on methodology. For example, as neural-based methods, they all use an architecture called Encoder-Decoder [1]. Next, With respect to the unique patching rate, we find that four NPR systems (CoCoNut [36], Recoder [67], SequenceR [5] and Tufano [54]) have a relatively higher rate of unique fixes than others, with respect to 27% (654 bugs), 39% (597 bugs), 24% (687 bugs) and 17% (188 bugs). We find that Recoder [67] is the one that has the highest unique patching rate. The possible reason is that Recoder [67] designs a special mechanism that directly edits the abstract syntax tree of the buggy program while other systems model the patch generation process as a textual generation of code tokens. Whatever, the observation on the unique rate indicates that searching for a way to incorporate percentages of various NPR systems will be a feasible schema to improve the performance of NPR.

*RQ2.2 Do NPR systems have a repair preference for bug types?*

**Table 5: Fix rates on bugs of different types. The second row represents the number of bugs of each type for evaluation.**

| NPR System | Simple Delete | Simple Replace | Simple Insert | Mixed |
|---|---|---|---|---|
| | 1,718 bugs | 2,078 bugs | 4,532 bugs | 4,487 bugs |
| CODIT | 3% | 1% | 0.4% | 0.2% |
| Edits | 3% | 2% | 1% | 1% |
| Tufano | 17% | 11% | 8% | 4% |
| CoCoNut | 40% | 27% | 17% | 8% |
| Recoder | 41% | 8% | 7% | 8% |
| SequenceR | 45% | 30% | 21% | 13% |
| Total | 75% | 42% | 31% | 21% |

*Method and Results.* We first count the fix rates of NPR systems on different-type bugs. Relying on the buggy code and the human-written patches provided by *NPR4J-Benchmark*, we divide the bugs into four categories from the perspective of needed editing operations: *Simple Delete*, *Simple Replace*, *Simple Insert* and *Mixed*. Each type represents the editing operation required to convert a bug to a patch. For example, a bug of type *Simple Delete* represents that the bug can be transformed to the patch by deleting some code tokens. *Mixed* means the bug need at least two types of edits to fix. Table 5 summaries our statistical results on NPR systems' fix rates of different-type bugs. As shown in the first row of the table, we first count the number of bugs belonging to four types on the main part of Diversity (12,815 bugs). The next rows present fix rates of NPR systems on each type.

*Finding 5.* We find that the current NPR systems perform great on generating code-removal patches but are poor at fixing more complex bugs. Concretely, We observe that among the four types of bugs, NPR systems does best on fixing ones that only requires deleting operations. CoCoNut [36], Recoder [67] and SequenceR [5] reach a fix rate over 40% on Simple Delete bugs. In total, we find 75% of Simple Delete bugs can be fixed by one of the six NPR system. Is this because there are more samples of Simple Delete type in the training data than other types? As we counted, the proportion of four types of bugs (14% for Simple Delete,20% for Simple Replace,37% for Simple Insert,29% for Mixed) in the training set is similar to that in the evaluation set. So we conclude that NPR systems really have a preference for repairing bugs of Simple Delete type. This is reasonable since replacing and inserting are more

**Table 6: Fix rates of NPR systems on bugs with different similarity. The similarity is calculated between one bug in the evaluation set and its nearest sample in the training set. For instance, in column 13, "= 1" and "448" denote that 448 bugs have identical buggy lines (but different contexts) with one sample in the training set. CODIT and Edits are excluded since they do not use a context beyond the buggy line.**

| NPR System | = 0 | <= 0.4 | <= 0.5 | <= 0.55 | <= 0.6 | <= 0.65 | <= 0.7 | <= 0.75 | <= 0.8 | <= 0.9 | < 1 | 1 | < 1 (%) | = 1 (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 263 | 363 | 1,053 | 1,906 | 3,427 | 5,283 | 7,112 | 8,757 | 10,134 | 11,772 | 12,367 | 448 | 12,367 | 448 |
| Tufano | 5 | 8 | 65 | 132 | 235 | 388 | 538 | 668 | 800 | 953 | 1,009 | 71 | 8% | 16% |
| Recoder | 0 | 1 | 55 | 155 | 343 | 565 | 831 | 1,050 | 1,227 | 1,420 | 1,497 | 41 | 12% | 9% |
| CoCoNut | 0 | 2 | 42 | 129 | 364 | 682 | 1,022 | 1,359 | 1,687 | 2,080 | 2,234 | 169 | 18% | 38% |
| SequenceR | 2 | 6 | 81 | 231 | 518 | 915 | 1,334 | 1,716 | 2,100 | 2,564 | 2,747 | 153 | 22% | 34% |

complex operations that require models to search extra tokens for fixing. We also observe that on the Mixed-type bugs that require at least two kinds of operations to generate the patches, performances of the NPR models are struggling. Even the best system SequenceR [5] can only reach a fix rate of 12.66%. In total, only 21% of Mixed-type bugs can be fixed by either one of the six NPR systems. It should also be concerned that the distribution of four types of bugs in the real-world may be not the same. Since the real distribution in the world is unable to measure, we take a look into bugs in *NPR4J-Benchmark*, which may somehow reflect the distribution. We find that there are 70% bugs with types of Simple Insert or Mixed. It seems that NPR systems still have a long way to go on fixing more complex bugs.

## 4.3 Generalizability

*RQ3 Can the NPR systems fix the bugs which have never been seen during training?*

*Method and Results.* Since we have already excluded all bug-fix pairs that appear in the evaluation set from the training data, there is no chance for models to see the bug and its fix scheme before fixing when evaluation. While in some cases, model may have seen the same buggy code lines under different contexts. **It should be emphasized that this is not a data leakage problem, because there are no same input-output pairs in the evaluation set as in the training set.** In order to measure to what extent the NPR system has already known about the bug before generating predictions, We calculated the similarity between the buggy line of each bug in the evaluation set (12,815 bugs in the diversity part) and its nearest sample in the training set. We use the similarity measuring algorithm coming from the difflib of python[8], using the funciton *difflib.SequenceMatcher.ratio()*. The algorithm calculates a measure of two sequences' similarity as a float in the range [0, 1] via the formula $2 * M/T$ where $M$ is the number of matches and $T$ is the total number of elements in both sequences. For example, "abcd" and "aefg" get a similarity of 0.25. Then, for each bug, we calculate its similarity between every bug of the 144,641 bugs that are used for training and record the nearest sample which has the biggest similarity with the bug. Next, we group 12,815 bugs in evaluation set according to its similarity with the nearest sample in the training set and count the fix rate of each NPR system in each group. We exclude CODIT [4] and Edits [8] since they do not use a context beyond the buggy line. The other four NPR systems use either a method-level context [36, 54, 67] or a class-level context[5].

[8]https://docs.python.org/3/library/difflib.html

*Finding 6.* NPR systems have the ability to fix bugs that have never been seen during training. As shown in Table 6, If the similarity of a bug and its nearest sample in the training set is less than 1, it means the NPR system has never seen an identical bug before. We find that NPR systems have good performances on such bugs. For example, SequenceR [5] reaches a fix rate of 22.21%, which is 98% performance on all bugs, according to the previous results on Figure 4. When lowing down the standard to less than 0.6 (as the difflib document reports, a ratio() value over 0.6 means the sequences are close matches), NPR systems still reach not bad performances, respectively reaching 39%, 18%, 81%, 85%, 56% and 67% of full performance. Even the bugs have nothing in common with the samples in the training data (the similarity is equal to 0), Tufano's model [54] and SequenceR [5] can still fix some of them. An interesting finding is that Tufano[54] fixes 5 and 8 bugs on "= 0" and "<= 0.4" parts, outperforming the other five NPR systems. We guess the possible reason is that Tufano[54] introduces an abstraction on the buggy code, replacing identifiers and variables with sequential ids. Thus, in some case, the model can have some knowledge of a "=0" bug after abstraction.

*Finding 7.* NPR systems have a higher probability to fix one bug if it has similar samples in the training data ("similar" refers to the similarity between bug lines). Taking a left-to-right view over Table 6, we observe that NPR systems always reach a relatively higher fix rate on bugs that have a higher similarity with the nearest sample in the training set. Specifically, when the NPR system has seen the identical buggy line during training (the similarity is equal to 1), the bug has a much more higher rate to be fixed correctly. Three systems (Tufano [54],CoCoNut [36], SequenceR [5]) have a significant performance improvement (54% to 200%) on samples with a similarity less than 1 and that equaling to 1. However, We find Recoder [67] is not sensitive to the similar sample, fixing 12% on samples with a similarity less than 1 and 9% on samples with a similarity equaling 1. This might be caused by its special representation on the source code. For each buggy method, Recoder [67] represents it as three inputs: (1) AST traversal sequence, (2) Tag embedding and (3 ) AST-based Graph. Such processed inputs may contain more structural information, but have less textual relationship with original codes. According to the above findings, collecting more data will be an effective way to improve the performance of

NPR systems, since a larger training set is more likely to contain samples that are similar to the bugs the systems are applied to fix.

## 5 THREATS TO VALIDITY

There are several threats to the validity of our study. The first threat is the hyperparameters of the six NPR systems. As is known, the same architecture model with different hyperparameters can yield very different performance. To mitigate this threat, we use a random search to make models trained to their best states. Second is the manual checking process performed in our experiment. In our experiment, to minimize the subjective deviation brought by inspectors, each test-adequate patch is checked by at least two of the authors and the reason why one is plausible or correct must be provided. Third is that our implementation of the *NPR4J-Benchmark* and *NPR4J-Framework* is unavoidable to have some bugs that may bring threats. We open source all our codes, data and experimental results for the verification of other researchers.

## 6 RELATED WORKS

Works related to ours are empirical studies on APR tools [13, 17, 23, 30, 33, 34, 38, 44, 46, 62, 64]. These studies focus on some empirical properties such as the efficiency of multiple APR systems. Among them, the most similar work is from Durieux et al.[9] that empirically validate 11 test-suite-based repair tools on 5 benchmarks. The authors run a huge experiment on 2,141 bugs and 23,551 repair attempts. They focus on the issue of *benchmark overfitting* and analyse the causes of non-patch generation.

Recently, we also notice there rises some analysis studies specifically for NPR [8, 45, 53, 58]. Tufano et al. [53] investigate the ability of a Neural Machine Translation (NMT) model to learn how to automatically apply code changes implemented by developers during pull requests. Ding et al. [8] and Namavar et al. [45] empirically compare advantages of different design on NPR such as context length and code tokenization.

Our study has two main differences compared with previous related works. First, to our best knowledge, this is the seminal study that performs huge empirical validation on multiple existing NPR systems. Second, we focus on excluding non-methodology threats and contributes a new benchmark and a framework tool.

## 7 CONCLUSION

In this paper, we focus on the empirical validation of six SOTA NPR systems. First, we identify three differences on setups of previous NPR systems that may bring threats to the validity of the comparison. To mitigate such threats, we construct a well-organized benchmark named *NPR4J-Benchmark* and a framework tool named *NPR4J-Framework* that supports training and evaluating NPR systems on a diversity of bugs. Based on the benchmark and framework, we perform a large-scale empirical experiment, training and evaluating six latest NPR system (Tufano [54], CoCoNut [36], CODIT [4], Edits [8], SequenceR [5] and Recoder [67]) on 13,706 java bugs under same setups.

As we investigated, NPR systems show great potential on fixing a diversity of bugs. The best system is able to repair 22% of the bugs in our evaluation set. However, we also find some shortcomings of NPR systems. For example, they are goot at dealing with code-delete bugs but poor at fixing complex bugs that require mixed editing operations. Our other findings point out several promising directions for follow-up works. First, we find that the six NPR systems share a not-low unique patching rate, which means searching for a way to combine advantages of various NPR systems could be useful for performance improving. Second, our findings on generalizability demonstrate that NPR systems have a higher probability to fix one bug if it has similar samples in the training data. In light of this finding, we conclude that collecting more samples for training is a practical trick to improve NPR.

## ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (61802167 and 62172214), Natural Science Foundation of Jiangsu Province, China (BK20201250 and BK20210279), Cooperation Fund of Huawei-NJU Creative Laboratory for the Next Programming, and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06). Chuanyi Li and Jidong Ge are the corresponding authors.

## REFERENCES

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.0473

[2] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin T. Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 780–791. http://proceedings.mlr.press/v139/berabi21a.html

[3] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. 2012. *Quantify the time and cost saved using reversible debuggers.* Technical Report. Technical report, Cambridge Judge Business School.

[4] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* (2020).

[5] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.* 47, 9 (2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[6] Zimin Chen and Martin Monperrus. 2018. The CodRep Machine Learning on Source Code Competition. *CoRR* abs/1807.03200 (2018). arXiv:1807.03200 http://arxiv.org/abs/1807.03200

[7] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.* OpenReview.net. https://openreview.net/forum?id=SJeqs6EFvB

[8] Yangruibo Ding, Baishakhi Ray, Premkumar T. Devanbu, and Vincent J. Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020.* IEEE, 275–286. https://doi.org/10.1145/3324884.3416587

[9] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: a large-scale experiment on 2, 141 bugs and 23, 551 repair attempts. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[10] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016*, Christof J. Budnik, Gordon Fraser, and Francesca Lonetti (Eds.). ACM, 85–91. https://doi.org/10.1145/2896921.2896931

[11] Thomas Durieux and Martin Monperrus. 2016. Introclassjava: A benchmark of 297 small and buggy java programs. (2016).

[12] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional Sequence to Sequence Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 1243–1252. http://proceedings.mlr.press/v70/gehring17a.html

[13] Davide Ginelli, Matias Martinez, Leonardo Mariani, and Martin Monperrus. 2020. A Comprehensive Study of Code-removal Patches in Automated Program Repair. *CoRR* abs/2012.06264 (2020). arXiv:2012.06264 https://arxiv.org/abs/2012.06264

[14] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning.* MIT Press. http://www.deeplearningbook.org/

[15] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[16] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder Singh and Shaul Markovitch (Eds.). AAAI Press, 1345–1351. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[17] Jiajun Jiang, Yingfei Xiong, and Xin Xia. 2019. A manual inspection of Defects4J bugs and its implications for automatic program repair. *Sci. China Inf. Sci.* 62, 10 (2019), 200102:1–200102:16. https://doi.org/10.1007/s11432-018-1465-6

[18] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[19] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 686–698. https://doi.org/10.1109/ICSE43902.2021.00069

[20] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[21] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[22] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[23] Xuan-Bach Dinh Le, David Lo, and Claire Le Goues. 2016. Empirical Study on Synthesis Engines for Semantics-Based Program Repair. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016.* IEEE Computer Society, 423–427. https://doi.org/10.1109/ICSME.2016.68

[24] Xuan-Bach Dinh Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1.* IEEE Computer Society, 213–224. https://doi.org/10.1109/SANER.2016.76

[25] Armin Lederer, Alexandre Capone, Jonas Umlauft, and Sandra Hirche. 2021. How Training Data Impacts Performance in Learning-Based Control. *IEEE Control. Syst. Lett.* 5, 3 (2021), 905–910. https://doi.org/10.1109/LCSYS.2020.3006725

[26] Suhua Lei, Huan Zhang, Ke Wang, and Zhendong Su. 2018. How training data affect the accuracy and robustness of neural networks for image classification. (2018).

[27] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 602–614. https://doi.org/10.1145/3377811.3380345

[28] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 162:1–162:30. https://doi.org/10.1145/3360588

[29] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. 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 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, Gail C. Murphy (Ed.). ACM, 55–56. https://doi.org/10.1145/3135932.3135941

[30] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019.* IEEE, 102–113. https://doi.org/10.1109/ICST.2019.00020

[31] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 456–467. https://doi.org/10.1109/SANER.2019.8667970

[32] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 31–42. https://doi.org/10.1145/3293882.3330577

[33] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A Critical Review on the Evaluation of Automated Program Repair Systems. *Journal of Systems and Software* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[34] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 615–627. https://doi.org/10.1145/3377811.3380338

[35] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 84:1–84:28. https://doi.org/10.1145/3133908

[36] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[37] Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 468–478. https://doi.org/10.1109/SANER.2019.8667991

[38] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empir. Softw. Eng.* 22, 4 (2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[39] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 441–444. https://doi.org/10.1145/2931037.2948705

[40] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, 65–86. https://doi.org/10.1007/978-3-319-99241-9_3

[41] Matias Martinez and Martin Monperrus. 2019. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *J. Syst. Softw.* 151 (2019), 65–80. https://doi.org/10.1016/j.jss.2019.01.069

[42] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021.* IEEE, 505–509. https://doi.org/10.1109/MSR52588.2021.00063

[43] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1 (2018), 17:1–17:24. https://doi.org/10.1145/3105906

[44] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs?. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 25. https://doi.org/10.1145/3180155.3182533

[45] Marjane Namavar, Noor Nashid, and Ali Mesbah. 2021. A Controlled Experiment of Different Code Representations for Learning-Based Bug Repair. *CoRR* abs/2110.14081 (2021). arXiv:2110.14081 https://arxiv.org/abs/2110.14081

[46] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *36th International*

*Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 254–265. https://doi.org/10.1145/2568225.2568254

[47] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world Java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 10–13. https://doi.org/10.1145/3196398.3196473

[48] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 648–659. https://doi.org/10.1109/ASE.2017.8115675

[49] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 1073–1083. https://doi.org/10.18653/v1/P17-1099

[50] Yu Tang, Long Zhou, Ambrosio Blanco, Shujie Liu, Furu Wei, Ming Zhou, and Muyun Yang. 2021. Grammar-Based Patches Generation for Automated Program Repair. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021 (Findings of ACL, Vol. ACL/IJCNLP 2021)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, 1300–1305. https://doi.org/10.18653/v1/2021.findings-acl.111

[51] TRICENTIS TEAM. 2017. $1.1 Trillion Impacted by Software Defects: A Testing Fail? https://www.tricentis.com/blog/1-1-trillion-in-assets-impacted-by-software-defects-a-software-testing-fail/

[52] Leonardo Trujillo, Omar M. Villanueva, and Daniel Eduardo Hernandez. 2021. A Novel Approach For Search-Based Program Repair. *IEEE Softw.* 38, 4 (2021), 36–42. https://doi.org/10.1109/MS.2021.3070552

[53] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 25–36. https://doi.org/10.1109/ICSE.2019.00021

[54] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4 (2019), 19:1–19:29. https://doi.org/10.1145/3340544

[55] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=ByloJ20qtm

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[57] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[58] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 479–490. https://doi.org/10.1109/SANER.2019.8668043

[59] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 416–426. https://doi.org/10.1109/ICSE.2017.45

[60] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 10799–10808. http://proceedings.mlr.press/v119/yasunaga20a.html

[61] Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 11941–11952. http://proceedings.mlr.press/v139/yasunaga21a.html

[62] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *J. Syst. Softw.* 171 (2021), 110825. https://doi.org/10.1016/j.jss.2020.110825

[63] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empir. Softw. Eng.* 26, 2 (2021), 20. https://doi.org/10.1007/s10664-020-09920-w

[64] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 740–751. https://doi.org/10.1145/3106237.3106262

[65] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Trans. Software Eng.* 46, 10 (2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648

[66] Wenkang Zhong, Chuanyi Li, Jidong Ge, and Bin Luo. 2022. Neural Program Repair: Systems, Challenges and Solutions. *arXiv preprint arXiv:2202.10868* (2022).

[67] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 341–353. https://doi.org/10.1145/3468264.3468544