J Inf Process Syst, Vol.18, No.5, pp.628~636, October 2022
https://doi.org/10.3745/JIPS.04.0251

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Applying Token Tagging to Augment Dataset for Automatic Program Repair

Huimin Hu and Byungjeong Lee*

Abstract

Automatic program repair (APR) techniques focus on automatically repairing bugs in programs and providing correct patches for developers, which have been investigated for decades. However, most studies have limitations in repairing complex bugs. To overcome these limitations, we developed an approach that augments datasets by utilizing token tagging and applying machine learning techniques for APR. First, to alleviate the data insufficiency problem, we augmented datasets by extracting all the methods (buggy and non-buggy methods) in the program source code and conducting token tagging on non-buggy methods. Second, we fed the preprocessed code into the model as an input for training. Finally, we evaluated the performance of the proposed approach by comparing it with the baselines. The results show that the proposed approach is efficient for augmenting datasets using token tagging and is promising for APR.

Keywords

Augment Dataset, Automatic Program Repair, Machine Learning, Token Tagging

1. Introduction

With the rapid development of information technology, software systems have been widely applied in various fields. Automatic program repair (APR) is becoming increasingly important for providing software with improved quality and reducing the burden on developers. APR aims to automatically repair bugs in programs with minimal or no human intervention.

APR is often associated with fault localization in obtaining buggy information. Based on program analysis techniques, fault localization determines buggy elements (such as buggy statements, methods, and files) and lists the ranked buggy elements and their corresponding suspicious scores. A suspicious score indicates the probability that a buggy element is a bug. The ranked list can be provided to developers to release their workload on manual debugging, and it can be provided to the APR. APR approaches repair bugs by utilizing the fault localization results, suggesting that the inaccuracy of bug localization can introduce noise to the APR. In many recent APR studies, researchers have assumed that fault localization results are perfect or have performed APR with externally provided perfect bug information. This minimizes the possible noise of the fault localization. Researchers have attempted to overcome these difficulties and have made significant progress in APR. However, most studies focused on repairing relatively simple bugs, such as single-line bugs and bugs with two or three consecutive lines.

Manuscript received August 23, 2022; accepted September 11, 2022.
* **Corresponding Author:** Byungjeong Lee (bjlee@uos.ac.kr)
Dept. of Computer Science, University of Seoul, Seoul, Korea (hyemin@uos.ac.kr, bjlee@uos.ac.kr)

In this study, we developed a novel APR approach that augments datasets to repair relatively complex bugs and performed model training on augmented datasets. In the proposed approach, we first preprocessed the program source code using token tagging to perform dataset augmentation. We used an augmented dataset for model training. After training, we evaluated the performance of the proposed approach by reporting the number of correctly repaired bugs and comparing the results with the baselines.

2. Motivation

2.1 Bug Categories

In this study, we divided the bugs into single-line and multi-line bugs. A single-line bug has only one buggy line and can be repaired by changing only one line. A multi-line bug means that repairing it requires changing several lines, or the bug is associated with many lines. In a bug, these lines may be continuous, broken, or a combination of both. Fig. 1 shows an example of the bug types.

```
 public static void main(String[] args) {
-    Long starttime = 0L;
+   Long starttime = System.currentTimeMillis();
     for(int i=0;i<10000;i++){
        logger.info("test"+i);
     }
     Long endtime = System.currentTimeMillis();
     System.out.println("running time : "+(endtime-starttime));
 }
```

a. A single-line bug

```
   public static void allFile(File f) {
         File[] files = f.listFiles();
-     for (File file : f) {
+     for (File file : files) {
          if (file.isDirectory()) {
                allFile(file);
          }else {
              String buggy = read(file);
-             String strs[] = buggy.split("");
-             write_tsv(file.getName(), strs.size());
+             String strs[] = buggy.split(" ");
+             write_tsv(file.getName(), strs.length);
             }
          }
    }
```

b. A multi-line bug (a combination of continuous and broken lines)

Fig. 1. Example of single-line bug and multi-line bug.

2.2 Why Multi-line Bugs?

To obtain more information regarding multi-line bugs, we investigated the distribution of the number of buggy lines in the Defects4J dataset [1]. Table 1 lists the number of single-line and multi-line bugs from Defects4J. For the project Chart, if an APR approach only focuses on repairing single-line bugs, it will miss 17 multi-line bugs. It does not make any effort to repair multi-line bugs. Consequently, it can repair a maximum of 9/26 bugs. In contrast, this opens up the possibility of repairing more bugs when no

line limits exist. Repairing relatively complex bugs, such as multi-line bugs, is a significant challenge that requires additional effort. In this study, the proposed approach focused on repairing single-line and multi-line bugs, indicating that it was possible to repair all 393 bugs.

Table 1. Number of single-line and multi-line bugs

	Chart	Closure	Lang	Math	Mockito	Time	All
Number of single-line bugs	9	26	13	24	8	3	83
Number of multi-line bugs	17	105	52	82	30	24	310
All	26	131	65	106	38	27	393

2.3 Why Augment Datasets?

Learning-based APR approaches require parallel data ("buggy-fixed" pairs); however, this type of data is insufficient [2]. Many researchers have collected data from websites to address the problem of insufficient data. Sometimes, the data were well organized and existed in the form of "buggy-fixed" pairs. However, the data are primarily raw source codes. Researchers can preprocess the raw source code as they prefer while facing the problem of insufficient information.

SemSeed [2] reports how to seed real bugs in programs. Gupta et al. [3] investigated mutant-based fault localization, and the basic objectives were to mutate the code snippets in the source code to observe the changes before and after mutation and to detect the effect of the mutation code snippets on the test results. Motivated by these two studies, we attempted to mutate the correct source code to generate a buggy code and make it appear like a real bug. In this way, we augmented datasets to help solve the insufficient data problem.

3. Proposed Approach

Fig. 2 shows the overall workflow of the proposed approach, which has three stages. In the data preparation stage (Section 3.1), we preprocessed the data using token tagging and organized the preprocessed "buggy-fixed" pairs into datasets. In the model training stage, we fed the prepared datasets as the input and trained a model called "LaserTagger" for patch generation. This is a model released by Google that can perform several natural language tasks. It should be noted that all types of errors contained in the training data (grammatical errors, logic errors, and others) are likely to be learned. In the patch validation stage (Section 3.2), we used two criteria to perform patch validation based on whether the datasets provided the test cases.

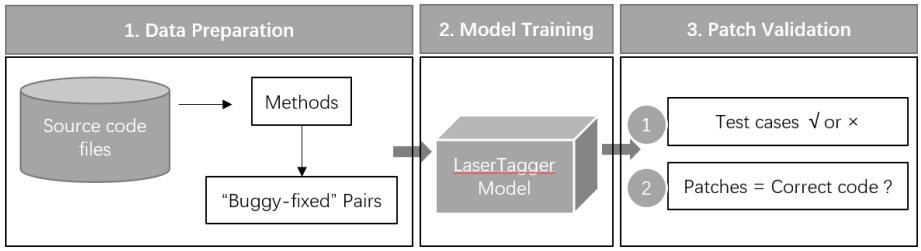

Fig. 2. Overall workflow of proposed approach.

3.1 Data Preparation

We augmented the datasets using the two steps shown in Fig. 3. First, we extracted all methods in the buggy files using a parser and classified them into buggy and non-buggy methods. Second, we organized these methods and their corresponding buggy or fixed versions into "buggy-fixed" pairs. As the original datasets were well organized, off-the-shelf corresponding fixed versions for the buggy methods existed, and we mapped them into a pair. However, no buggy version for non-buggy methods existed. We used token tagging to artificially generate buggy versions. These "token tagging methods and non-buggy methods" pairs are called token tagging method pairs. The final augmented datasets contained the method pairs from the original datasets and the token tagging method pairs.

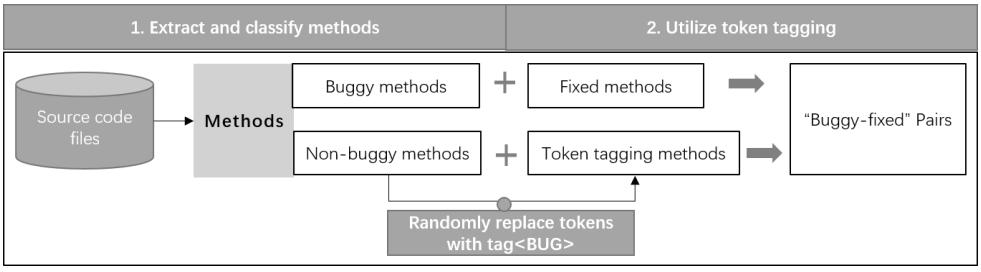

Fig. 3. Augmentation of datasets using token tagging.

We replaced the tokens of random positions and numbers of replacements in the non-buggy methods with tag <BUG>. Furthermore, to minimize the effect of code form errors (such as missing semicolons and incorrect parentheses) on model learning, we avoided tokens related to the code form when replacing tokens. Fig. 4 shows a sample for creating a buggy version for a non-buggy method. Replacing several positions makes the code resemble a multi-line bug. Using these data for model training may enable the model to handle more complex bugs.

```
public static Logger get(String
   logName){ if (logName != null){
      return loggerMap.get(logName);
   } return null;
}
```

a. A non-buggy method

```
public static Logger get(<BUG>
   logName){ if (logName != null){
      return loggerMap.<BUG>(logName);
   } return null;
}
```

b. The corresponding buggy method with <BUG>s

Fig. 4. Non-buggy code and its token tagging buggy version.

3.2 Patch Validation

For datasets containing test cases, we validated the generated patches by checking whether they could pass the test cases. In Eq. (1), $P - value$ ranges between 0 and 1. When $P - value$ is 1, it means that the

current project passed all the test cases, and all the patches in it are plausible. The plausible patches, which have the same meaning as the human-written patches, are correct. In this paper, we report the experimental results by showing the number of plausible and correct patches at the project level.

$$P - value = \frac{Num_{passed\_test\_cases}}{Num_{all\_test\_cases}} \quad (1)$$

For the dataset without test cases, we checked the number of patches corresponding to the correct source code and calculated the accuracy. In Eq. (2), $Num_{matches}$ is the number of patches that is the same as the correct source code; these patches are regarded as correct patches.

$Num_{patches}$ is the total number of all the generated patches. This criterion may miss some semantically correct patches. As the reference information is insufficient, we have not manually confirmed the patches that do not precisely match the correct code but may be correct.

$$Accuracy = \frac{Num_{matches}}{Num_{patches}} \quad (2)$$

4. Experiments

4.1 Datasets

We conducted experiments on datasets Bugs2Fix [4] (without test cases) and Defects4J [1] (with test cases), both of which contained single-line and multi-line bugs. Bugs2Fix was used for model training and patch validation, whereas Defects4J was used only for patch validation. In all the experiments, we utilized the perfect bug information provided by the datasets.

Bugs2Fix is a large-scale dataset. Some source code file pairs were processed into "buggy-fixed" method pairs. As listed in Table 2, the data in "buggy-fixed" method pairs have two versions: small with 58,350 pairs and medium with 65,455 pairs. Dataset augmentation was performed based on these two datasets. We set the number of token tagging data to the same size as the original, which may help reduce the noise and bias that the token tagging data might introduce into the datasets. As the original training set had 46,680 data points in the small dataset, we generated 46,680 token tagging data as the training data. In this manner, the small augmented dataset contained 93,360 training data. Furthermore, for comparison with other studies, we used the original test set in the datasets of the augmented versions.

Table 2. Bugs2Fix datasets with "buggy-fixed" method pairs

	Small	**Small$_{augment}$**	**Medium**	**Medium$_{augment}$**
Train	46,680	93,360	52,365	104,730
Valid	5,835	11,670	6,545	13,090
Test	5,835	5,835	6,545	6,545
All	58,350	110,865	65,455	124,365

Defects4J includes several projects, six of which are widely used in APR research. We analyzed Defects4J at three levels (project, file, and method levels) in terms of the number of bugs (Table 3). At the project level, a buggy project may contain buggy and non-buggy files. The buggy files may be one

or more. At the file level, a buggy file contains buggy and non-buggy methods. The buggy methods in a Java file can also be one or more. Defects4J provides test cases at the project level that can be run directly according to the application provided by the developer.

Table 3. Bugs in Defect4J

	Number of buggy projects	Number of buggy files	Number of buggy methods
Chart	26	28	39
Closure	131	143	176
Lang	65	65	90
Math	106	119	139
Mockito	38	46	89
Time	27	30	48
All	393	431	581

4.2 Baselines

To investigate the performance of the proposed approach, we compared it with that of Tufano et al. [5]. They abstracted buggy and fixed methods, tagged the customized method names and variable names to obtain unique IDs, and recorded a mapping for them. For example, it replaces the customized method name, "addNodes," with <METHOD_1>, and records a mapping ["addNodes," <METHOD_1>]. The corresponding tags must be converted into the original tokens after generating patches, and it is difficult to use method and variable names missing in the corresponding records.

To investigate whether the LaserTagger model is efficient, we compared it with three models by performing experiments on the original Bug2Fix datasets. Long short-term memory (LSTM) [6], a unique recurrent neural network, mainly solves the problems of gradient disappearance and explosion during long-sequence training. Transformer [7] was developed by Google's team in 2017. It uses a self-attention mechanism, can be trained in parallel, and can provide global information. CodeBERT [8] is based on a transformer-based neural architecture trained with a hybrid objective function, including the replaced token detection pretraining task.

4.3 Research Questions

RQ1: How does the proposed approach perform?

RQ2: What is the reason for the different performances of the proposed approach and baselines?

4.4 Performance of Proposed Approach

We compared the results of the LaserTagger model and proposed approach with those of previous studies [5, 9] (Table 4). We obtained the results of LSTM, Transformer, and CodeBERT from Codexglue [9] and the results of Tufano et al. [5] from their published paper. We set the experiments as Codexglue: generate one patch for each bug and use the data originally released from the Bugs2Fix datasets.

In the models part, LaserTagger achieved better results than the other three baseline models for the small dataset, and CodeBERT was better for the medium dataset (Table 4). However, this does not diminish LaserTagger's potential for APR. The study showed that the proposed approach correctly repaired 19.76% and 5.41% of the bugs in the small and medium datasets, respectively. Hence, the

approach proposed in this study performs better than that of Tufano et al. [5].

Furthermore, the difference between the LaserTagger model and the proposed approach is that the latter uses token tagging, unlike the former. Based on the results, using token tagging to perform dataset augmentation is meaningful. This alleviates the insufficient dataset problem and can be used to address inaccurate bug information.

We also performed bug repairs using Defects4J. The proposed approach correctly repaired 14 bugs at the project level (Table 5). C, CL, L, M, and Mo denote Chart, Closure, Lang, Math, and Mockito, respectively.

Table 4. Results for Bugs2Fix datasets

			ACC_{small} (%)	ACC_{medium} (%)
Model	Original[a]	LSTM	10	2.5
		Transformer	14.7	3.7
		CodeBERT	16.4	**5.16**
		LaserTagger	**17.82**	4.74
Study	Unique IDs[b]	Tufano et al. [5]	9.22	3.22
	Token tagging[c]	Proposed approach	**19.76**	**5.41**

[a]The small and medium datasets were originally released from Bugs2Fix [5].
[b]The datasets were preprocessed by replacing tokens in the original dataset with unique IDs.
[c]The datasets were augmented by applying token tagging on the original.

Table 5. Results on Defects4J dataset

Category	Bug IDs	# Bugs
Single-line bugs	C-24, CL-18, 38, L-26, 57, M-2, 11, 75, Mo-38	9
Multi-line bugs	C-14, L-55, M-20, 22, 67	5

4.4.1 Answer to RQ1: Performance of proposed approach

The proposed approach correctly repaired 19.76% of the bugs in the Bugs2Fix small dataset, 5.41% in the medium dataset, and 14 bugs in Defects4J. The results show that the LaserTagger model is a reasonable augmenting dataset for APR. Second, the proposed approach performs better than that of Tufano et al. [5]. Finally, the proposed approach correctly repaired five multi-line bugs in Defects4J, enhancing the possibility of repairing more multi-line bugs.

4.4.2 Answer to RQ2: Reason for different performances

By comparing Tufano et al.'s [5] approach with LSTM, Transformer, and CodeBERT, we observed that with the same experimental setting, their approach negatively influenced the performance. As mentioned in Section 4.2, their work required converting unique IDs into original tokens. It is challenging to handle tokens not included in the corresponding records. This means that Tufano et al.'s [5] approach requires sufficient records of method/variable names and their corresponding unique IDs. Otherwise, performance is affected. Their approach did not obtain sufficient records and failed to revert some newly generated method names and variable names. However, the proposed approach performs augmentation on the same datasets and achieves better results.

5. Conclusion

In this study, we developed an approach that augments datasets by utilizing token tagging for automatic bug repair. First, we augmented the datasets using token tagging. We tokenized the source code and randomly replaced the tokens with the correct source code to create buggy versions. We then used the preprocessed data as the input for model training and validated the generated patches by checking whether the patches matched the fixed versions in Bug2Fix and executing test cases in Defects4J. We observed the number of methods in which the patch and the fixed code were consistent in Bugs2Fix and the number of correctly repaired projects in Defects4J. We also compared the performance of the proposed approach with that of the baselines and analyzed the results. In the future, we plan to improve this approach by reviewing more studies on generating datasets and code mutants and augmenting datasets by creating more real-like bugs. Furthermore, we will perform more analyses on various datasets to improve the multi-line bugs, particularly by analyzing additional practical programs to enable the proposed approach to handle more real bugs.

Acknowledgement

This work was supported by National Research Foundation of Korea (NRF) grants funded by the Korean government (MSIT) (No. NRF-2020R1A2B5B01002467 and NRF-2022M3J6A1084845).

References

[1] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: a database of existing faults to enable controlled testing studies for Java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, 2014, pp. 437-440.

[2] J. Patra and M. Pradel, “Semantic bug seeding: a learning-based approach for creating realistic bugs,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, 2021, pp. 906-918.

[3] N. Gupta, A. Sharma, and M. K. Pachariya, “A novel approach for mutant diversity-based fault localization: DAM-FL,” *International Journal of Computers and Applications*, vol. 43, no. 8, pp. 804, 2021.

[4] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “An empirical investigation into learning bug-fixing patches in the wild via neural machine translation,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier, France, 2018, pp. 832-837.

[5] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 4, article no. 19, 2019. https://doi.org/10.1145/3340544

[6] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in Neural Information Processing Systems*, vol. 30, pp. 5998-6008, 2017.

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, et al., "CodeBERT: a pre-trained model for programming and natural languages," 2020 [Online]. Available: https://arxiv.org/abs/2002.08155.

[9] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, et al., "Codexglue: a machine learning benchmark dataset for code understanding and generation," 2021 [Online]. Available: https://arxiv.org/abs/2102.04664.

Huimin Hu **https://orcid.org/0000-0002-1470-0839**

She received her B.S. and M.S. degrees in Computer Science from Xinzhou Teachers University and the University of Seoul in 2019 and 2022, respectively. Since September 2022, she has worked as a PhD Candidate at the University of Stuttgart. Her current research interests mainly focus on improving the quality and efficiency of software.

Byungjeong Lee **https://orcid.org/0000-0002-2750-7608**

He received his B.S., M.S., and Ph.D. degrees in Computer Science from Seoul National University in 1990, 1998, and 2002, respectively. He was a researcher with Hyundai Electronics, Corp. from 1990 to 1998. Currently, he is a professor in the Department of Computer Science and Engineering at the University of Seoul, Korea. His research areas include software engineering and machine learning.