Code Naturalness to Assist Search Space Exploration in Search-Based Program Repair Methods

Altino Dantas[1(✉)], Eduardo F. de Souza[1], Jerffeson Souza[2], and Celso G. Camilo-Junior[1]

[1] Intelligence for Software Group, Federal University of Goiás, Alameda Palmeiras, Quadra D, Câmpus Samambaia, Goiânia 74690-900, Brazil
{altinobasilio,eduardosouza,celso}@inf.ufg.br

[2] Optimization in Software Engineering Group, State University of Ceará, Doutor Silas Munguba Avenue, 1700, Fortaleza 60714-903, Brazil
jerffeson.souza@uece.br
http://i4soft.com.br

Abstract. Automated Program Repair (APR) is a research field that has recently gained attention due to its advances in proposing methods to fix buggy programs without human intervention. Search-Based Program Repair methods have difficulties to traverse the search space, mainly, because it is challenging and costly to evaluate each variant. Therefore, aiming to improve each program's variant evaluation through providing more information to the fitness function, we propose the combination of two techniques, Doc2vec and LSTM, to capture high-level differences among variants and to capture the dependence between source code statements in the fault localization region. The experiments performed with the IntroClass benchmark show that our approach captures differences between variants according to the level of changes they received, and the resulting information is useful to balance the search between the exploration and exploitation steps. Besides, the proposal might be promising to filter program variants that are adequate to the suspicious portion of the code.

Keywords: Automated Program Repair · Search space exploration · Code naturalness

1 Introduction

Automated Program Repair (APR) is a research field that aims to fix buggy code without human intervention. Search-Based Program Repair algorithms [1] are based on the generate-and-validate approach, where variations of the original (bugged) code are generated and then evaluated, mostly, by a test suite. This evaluation method is time-consuming, might lead the search to plateaus or local

© Springer Nature Switzerland AG 2019
S. Nejati and G. Gay (Eds.): SSBSE 2019, LNCS 11664, pp. 164–170, 2019.
https://doi.org/10.1007/978-3-030-27455-9_12

optimal, and does not provide enough information to the fitness function to accurately differentiate the variants.

There is evidence that using dynamic analysis to capture internal states of the program and then using this information in the fitness function helps to differentiate variants better and improves the search expressiveness [2]. Although promising, this technique is still too costly and given that the search space is usually densely populated by plausible or low-quality solutions, there is still the need for efficient methods. Other approaches using neural network models to evaluate or classify source code in the APR context [3,4] might be used to help to compose the fitness function, but there is no evidence of a relation of those approaches to a fitness evaluation.

Previous research acknowledges that source code has similar properties of natural language and, therefore, the models used to compare natural language suits this new context [5]. Although [6] points out that such models might not be accurate, this research goes one step further and provide an efficient program's variant evaluation through the combination of two techniques, Doc2vec and LSTM, to capture high-level differences among variants (high-order mutations) and to capture the dependence between source code statements in the fault localization region (low-order mutation).

2 Proposed Approach

End-to-end search-based Automated Program Repair techniques comprise several steps with complex tasks involved. Yet, the focus of this proposal is about traversing the program's landscape concerns. Figure 1 presents an overview illustrating what APR aspects are involved with and are impacted by this proposal.

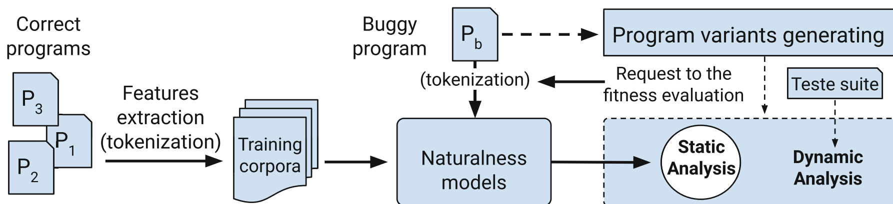

Fig. 1. Our proposal of using code naturalness for Automated Program Repair.

Our approach considers that correct source codes (written in the same programming language as the buggy code) are available in order to perform an automated program repair. Assuming that language models are trained to capture naturalness from scriptures, we rely on that to also assume that they are capable of capturing naturalness from correct source code. The corpus used to train the models are composed of tokens obtained through feature extraction from the correct source codes.

One can use information from the naturalness model in a traditional search-based program repair workflow (dashed lines). As Fig. 1 highlights with the white circle, we propose to use those models to help as a static analysis factor to a fitness function. The model receives a tokenized source code, transformed with the same process as used in training, and it returns information to compound the fitness along with the dynamic factor, which is typically based on a test suite and requires executing the program.

Among the existing ways to capture naturalness information from a corpus, we propose the usage of Doc2vec and LSTM. The motivation to use them, as well as their required inputs and the expected output, is detailed next.

2.1 Doc2vec Model

Doc2vec is an unsupervised machine learning method that learns a fixed feature representation to describe paragraphs and documents [7] throughout the distributions of words and sentences in a corpus. Previous work has proposed metrics to evaluate program variants based on a word embedding [3], but we chose the Doc2vec because it is more appropriate to deal with the whole document.

We employ Doc2vec to encode methods or functions in source codes in the same way it treats paragraphs in documents; thus, our technique obtains a measure of similarity between programs based on their vector representation. Given this information, one might assume that a fix is not so distant from the original bugged code. Besides, one can use this vector representation to investigate the impact of mutation operators, which is typically hard to do using the patch or AST representations.

We developed our proposal upon the Doc2vec provided by the Gensim[1] library in such a way that our model, when trained, receives two source codes and returns how similar they are.

2.2 LSTM Model

Long Short-Term Memory (LSTM) [7] is a recurrent neural network applied to pattern recognition in several contexts as text sequences, temporal series, genomes, and spoken words. It deals with sequences of elements without a regular interval of dependence between them. A similar situation occurs in source codes.

Different from analyzing the whole file, likewise the previous Doc2vec model, we propose to use LSTM to capture data from and generate information for a specific area of the code. Therefore, we train our LSTM model (implemented with the TensorFlow framework[2]) on the corpus of correct programs and then use it to synthesize code patch considering a part of the buggy program as the input sequence. The model receives a sequence of tokens from the suspicious region and generates the sequence it considers more "natural" for that part of the code. The experiment section presents how such an output sequence can be used to evaluate program variants.

---

[1] https://radimrehurek.com/gensim.
[2] https://www.tensorflow.org.

3 Preliminary Empirical Study

We conducted our experiments to answer the following Research Question: **Do naturalness models provide useful information to explore a program's search space?**

Thus, to evaluate our proposal, we used the IntroClass[3] benchmark. Considering this benchmark presents buggy and fixed versions but does not have fault localization data nor the correct patches, we first performed an inspection to generate such data. From the 99 available versions, distributed in six categories (problems), we selected 70 of them. We did not consider fixes achieved only by deletion, changes on the whole file, or empty diff between the bugged and fixed versions.

The tokenization step is as follows: For all 70 versions selected (140, taking the buggy and the correct ones), we removed all headers and comments, separated every relevant token by space (e.g., *vector[i]* became *vector [i]*) and replaced strings with the token "STRRPL" to prevent noise by words unlike to the one from the programming language. Thus, the tokenization process is a feature extraction process in the sense that each token represents a feature. After that, the proposal trained both Doc2vec and LSTM models over the corpora from the correct codes.

Three versions were randomly selected from each problem to test the former model. For each version, 15 variants were generated by applying GenProg's [8] mutation operators and the number of mutations (1, 2, 3) they received clustered those variants. Thus, it was generated 270 variants, and then it was possible to calculate the similarity between the original version and its variants. Such a metric is a unit percentage [0,1].

To the latter, the information of fault localization and a correct patch for all 70 versions were used to process the test and compute the accuracy (Acc) and precision (Prec) metrics. Acc is $\frac{|T \bigcap Y|}{|T|}$ and Prec is $\frac{|T \bigcap Y|}{|Y|}$, where T is the set of tokens in the knowing patch fix, and Y is the set of tokens predicted by the model. For this model requires an input sequence of tokens to produce another sequence, different configurations of length for both sequences were verified.

Although training such models might be expensive, it occurs only once, and its results are then used in $\mathcal{O}(1)$ time. A search-based program repair technique could benefit from this while using the information from the models to pre-evaluate a variant, alleviating the time-consuming process of running the test suite.

Experiments' data, scripts, and raw results are available at: https://altinodantas.github.io/sbpr-naturalness.

3.1 Preliminary Results

Figure 2 shows the average similarity grouped by the number of mutations the variants received and the problem they were implemented for. In some cases,

[3] https://repairbenchmarks.cs.umass.edu.

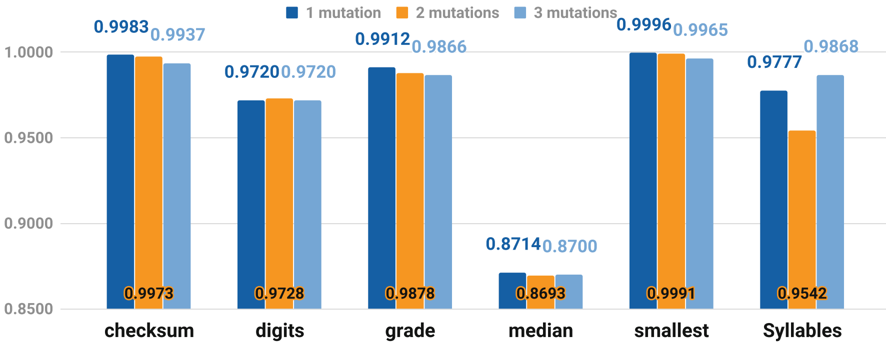

Fig. 2. Similarity between original buggy versions and their variants.

such as *checksum*, *grade*, and *smallest*, the higher the number of mutations, the lower the similarity. This behavior occurs if we assume that applying more perturbations increases the entropy between the original and a variant from it. However, for other problems, fewer mutations produced a lower similarity.

While inspecting the variants, we observed that in some cases, one mutation could have more impact than three mutations. For instance, in the *syllables* problem, a one-mutation variant deleted a "**for**" statement that had the main functionality of the program. Meanwhile, for the same version, a three-mutations variant computed higher similarity because the changes they provoked were not as profound.

This observation is exciting because it provides a method to capture the impact caused by mutation operators without the need of running the variant or the original program against the test suite, which is a time-consuming task. Therefore, a search algorithm could use the similarity to balance the exploration and exploitation, which are two crucial steps to cover a search space adequately. For example, using a threshold on the similarity values, one may enable mutations with more or less impact on the code.

Looking at the reports from the *median* problem, one notices that their values are inferior to the others. We speculate that this behavior is because the *median* has more versions than the other problems. Thus, with more information from the *median* to perform the training phase, the resulting model is more sensible to get differences between versions from that problem. However, a more rigorous investigation is needed.

Moving to the LSTM model evaluation, Table 1 presents the average of Acc and Prec for each configuration considering all versions and problems. Configurations have the format Xin_Yout, where X and Y indicate the number of tokens given as input and number of tokens expected in the output, respectively. As Acc and Prec may be conflicting, the results show, in some sense, which configurations present the best trade-off.

It is possible to notice the configurations with only five tokens in the input, the left part of the table, do not achieve the best values in Acc or Prec.

Table 1. Average of Acc and Prec achieved by LSTM model in all 70 selected versions.

Configuration	Acc	Prec	Configuration	Acc	Prec	Configuration	Acc	Prec
5in_10out	0.25	0.29	10in_10out	0.37	**0.35**	15in_10out	0.35	0.33
5in_20out	0.35	0.28	10in_20out	0.44	0.26	15in_20out	0.43	0.26
5in_30out	0.37	0.24	10in_30out	0.44	0.22	15in_30out	**0.45**	0.20

We might conclude that, on average, for the IntroClass' problems, five tokens are not enough to infer the dependence context due to some statements that are related to others further away.

Rather, since Acc and Prec's trade-off, all the 10-input configurations are not dominated, that is, there is no other configuration with results at least equal to one metric and strictly superior in the other. Notice that 10in_10out achieved the best Prec (0.35). Despite 15in_30out reached the best Acc (0.45) and also may be considered non-dominated, ten tokens to the input seem to be sufficient because increasing the input does not necessarily achieve better accuracy, as the others two 15-input does not overcome the best 10-input in this metric.

It is clear that based on local naturalness, given the faulty location and a correct patch, it was impossible to predict all the tokens needed to a fix. However, at least 37% of those tokens are always found in non-dominated settings; thus, this information could be used to discard variants that present fewer tokens than the model predicts. For instance, one could infer a threshold by analyzing the accuracy of the model over known fixed codes. This makes sense once some mutations are more suitable regarding the region of code they are applied.

Finally, considering the findings presented in this section, we can answer the Research Question saying that: "**yes, naturalness models can provide useful information to be employed by a technique that needs to explore a program's search space**". From the Doc2vec model, it is possible to get information to control the exploration and exploitation, and from the LSTM it is possible to create a filter on variants that are more suitable to the context of the region pointed by the fault localization.

4 Threats to Validity

The machine learning methods used to infer the naturalness models have a stochastic nature; thus, we performed preliminary training to get the model and decide about the hyperparameters. Nevertheless, fine-tuning the training process may generate different results. Since the investigated benchmark has small C programs, we can not generalize our findings to another programming language. However, previous work [4] presented evidence that LSTM works for real-world programs, including different programming languages. Finally, the results are hugely dependent on the tokenization we adopted.

5 Final Remarks

Several APR methods fix buggy programs by generating and validating variants. However, exploring a program's search space continues to be challenging. Therefore, this paper introduced an approach to generate useful information to explore a program's search space from the naturalness of correct programs. Preliminary results showed the proposal could potentially help APR methods to control their exploration and exploitation steps and filter variants regarding the fault localization data.

Next, we intend to couple our proposal to a search-based APR method. For that, we are working on integrating it to GenProg. Our models will then be used to prevent executing not such promising variants. Then, they will be used to compound the fitness function itself.

Acknowledgements. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and by the Fundação de Amparo à Pesquisa de Goiás (FAPEG).

References

1. Harman, M., Jones, B.F.: Search-based software engineering. Inf. Softw. Technol. **43**(14), 833–839 (2001)
2. de Souza, E.F., Le Goues, C., Camilo-Junior, C.G.: A novel fitness function for automated program repair based on source code checkpoints. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, pp. 1443–1450. ACM, New York (2018)
3. Amorim, L.A., Freitas, M.F., Dantas, A., de Souza, E.F., Camilo-Junior, C.G., Martins, W.S.: A new word embedding approach to evaluate potential fixes for automated program repair. In: Proceeding of the 2018 International Joint Conference on Neural Networks (IJCNN), pp. 1–8, July 2018
4. Roque, L., Dantas, A., Camilo-Junior, C.G.: Programming style analysis with recurrent neural network to automatic pull request approval. In: Proceedings of The 2019 International Joint Conference on Neural Networks (IJCNN). ijcnn.org (2019, to be appear)
5. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering, ICSE 2012, pp. 837–847. IEEE Press, Piscataway (2012)
6. Jimenez, M., Checkam,T.T., Cordy, M., Papadakis, M., Kintis, M., Le Traon, Y., Harman, M.: Are mutants really natural?: A study on how naturalness helps mutant selection. In: Proceedings of the 12th ESEM, page 3. ACM, 2018
7. Le, Q., Mikolov, T.: Distributed representations of sentences and documents. In: Proceedings of the 31st International Conference on Machine Learning, vol. 32, ICML 2014, pp. II-1188–II-1196. JMLR.org (2014)
8. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automatic software repair. IEEE Trans. Software Eng. **38**(1), 54–72 (2012)