Utilizing Source Code Embeddings to Identify Correct Patches

Viktor Csuvik*, Dániel Horváth†, Ferenc Horváth*, László Vidács*†

* *Department of Software Engineering*
†*MTA-SZTE Research Group on Artificial Intelligence*
University of Szeged, Szeged, Hungary

{csuvikv,skyzip,hferenc,lac}@inf.u-szeged.hu

***Abstract*—The so called Generate-and-Validate approach of Automatic Program Repair consists of two main activities, the generate activity, which produces candidate solutions to the problem, and the validate activity, which checks the correctness of the generated solutions. The latter however might not give a reliable result, since most of the techniques establish the correctness of the solutions by (re-)running the available test cases. A program is marked as a possible fix, if it passes all the available test cases. Although tests can be run automatically, in real life applications the problem of over- and underfitting often occurs, resulting in inadequate patches. At this point manual investigation of repair candidates is needed although they passed the tests. Our goal is to investigate ways to predict correct patches. The core idea is to exploit textual and structural similarity between the original (buggy) program and the generated patches. To do so we apply Doc2vec and Bert embedding methods on source code. So far APR tools generate mostly one-line fixes, leaving most of the original source code intact. Our observation was, that patches which bring in new variables, make larger changes in the code are usually the incorrect ones. The proposed approach was evaluated on the QuixBugs dataset consisting of 40 bugs and fixes belonging to them. Our approach successfully filtered out 45% of the incorrect patches.**

***Index Terms*—automatic program repair, apr, code embeddings, doc2vec, bert**

I. Introduction

Recently, a large number of Automatic Program Repair (APR) tools have been proposed [1]–[7] and many of these follow the Generate-and-Validate (G&V) approach. This approach first localizes the bug in the source code, typically with testing and ranking instructions based on how *"suspicious"* they are. The intuition is that if the suspicious parts are repaired, the program will work correctly. After localization (usually by search-based methods), patch candidates are generated and then validated, typically by (re-)running the tests. A program is marked as a possible fix, if it passes all the available test cases. This latter condition gives no assurance that the program is *correct*, since over- and underfitting [8] often occurs, resulting in inadequate patches. For programs that pass most tests, the tools are as likely to break tests as to fix them [9].

To overcome these limitations several approaches were introduced lately [10]–[13]. Although our work is related to these, it is a bit different. Let us consider a software system with a bug in it. With current state-of-the APR tools one could generate candidate patches, and get one or more potentially fixed programs. Since overly abundant recommendations can diminish the usefulness of the information itself, we try to define an ordering between candidate patches, as seen in Figure 1, while keeping the technique as simple as possible.

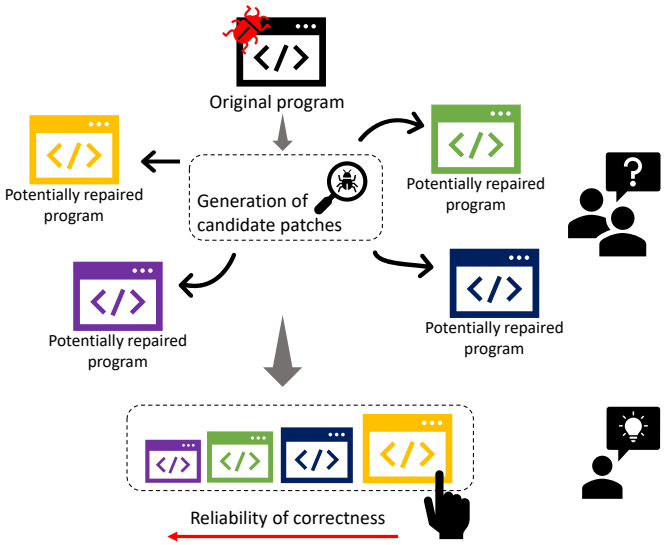

Fig. 1. A high level illustration of the patch validation process

Thus, our main motivation is to give developers a hint, which repair candidate to check first. To do so, textual and structural similarity was measured between the original (buggy) program and the generated patches. Our assumption was that the *correct* program is more similar to the original one, than other candidates. It comes from the perception that the current techniques mostly create one-line code modifications, thus leaving most of the original source code intact. Since both correct and incorrect patches by APRs have this property about one-line code modification, this reasoning is still possibly not convincing enough. However when repairing a program, it is preferable to construct patches which are simple and readable. This is because responsible software maintainers would review and inspect a patch carefully before

accepting it [14] – which occurs only if they judge that the patch is correct and safe [15]. Although textual and structural similarity does not imply, that the constructed patch is *simple* for human software developers, we found that, in some way, similarity indicates understandability and if a patch is more understandable, its chance of being *correct* is higher. To the best of our knowledge, most of the existing automatic repair tools do not explicitly take into account of the simplicity of a patch, however, there are exceptions such as DirectFix [15].

The source code similarities are measured using document/sentence embeddings, specifically with two state of the art techniques borrowed from the natural language processing domain: Doc2vec [16] and Bert [17]. Deciding the correctness of a candidate patch [18], [19] is one of the current challenges in automatic program repair, and today considered as an open question [20]. To our best knowledge this is the first study tackling the problem using embedding methods. However, there are other state-of-the-art repair patch ranking techniques [10], [13], [21], [22], which does not make use of source code embeddings.

In this paper we provide experimental results measured on the QuickBugs dataset [23]. We found that in principle similarity-based techniques can identify incorrect patches, in our dataset 45% of them were filtered out.

The paper is organized as follows. In the next section we introduce our research questions. Next high-level overview of our research is presented in Section III by depicting the proposed approach to order candidate repairs by their reliability. Next, we introduce the examined database and its representations upon which the experiments were carried out. Evaluation and analysis are presented in Section IV. Related work is discussed in Section V, and we conclude the paper in the last section.

II. Research Objectives

To investigate the benefits of the document/sentence embedding approach we organize our experiment along the following research questions:

RQ1: *What are the main differences of the similarities produced by the assessed embedding techniques?*

RQ2: *How embeddings learned on various source code representations perform compared to each other?*

RQ3: *Can similarity-based patch validation filter out incorrect patches?*

III. Methods

To measure the reliability of different patches, and to define an ordering between them, two different source code embedding methods were employed. These embedding methods, namely Doc2vec and Bert, operated on three different source code representations: SRC, AST and IDENT. In this section we describe the applied procedure.

Figure 2 shows the comprehensive approach we propose. First, from the original program the candidate patches are generated using external APR tools. From the original program and from the generated potentially fixed programs, we extract the three representations. For every representation we train the Doc2vec model separately, thus each built model is different. In case of Bert no training is required, since we used a pre-trained version of it. Regardless of the embedding method, for a code snippet an N dimensional vector is created on which one can measure similarity. These vectors contain information about the meaning, environment, and context of a word or document. The basic assumption is that the correct patches are more similar to the original program than the incorrect ones. Finally, from the learned similarities a ranked list is created: the first patch in this list is the *most similar* to the original program, thus by our assumption it is the most reliable.

A. Data Collection

We designed our approach to be applicable to projects written in Java, but in general the featured technique is independent of programming languages, since it only leans on the source code (text files). Even so the programming language is arbitrary, data collection proved to be a difficult task. Firstly, a publicly available database of bugs (and their fixes) was required, on the other hand, these patches should be revised by humans, to determine if they are correct or not. The QuixBugs benchmark [23] is a database having the purpose to objectively compare the currently most popular software repair tools. The database contains 40 single-line bugs for Python and Java, for which patches have been generated with the following tools: Arja, Cardumen, JGenprog, JKali, Nopol, RsRepair, NPEfix, Tibra and Mutation. Out of these 40 bugs, the listed tools managed to generate a total of 64 candidate patches for 14 distinct bugs. This means that there is an average of 4.5 patches per bug, each one requiring manual validation. In the database the generated patches are marked based on their correctness.

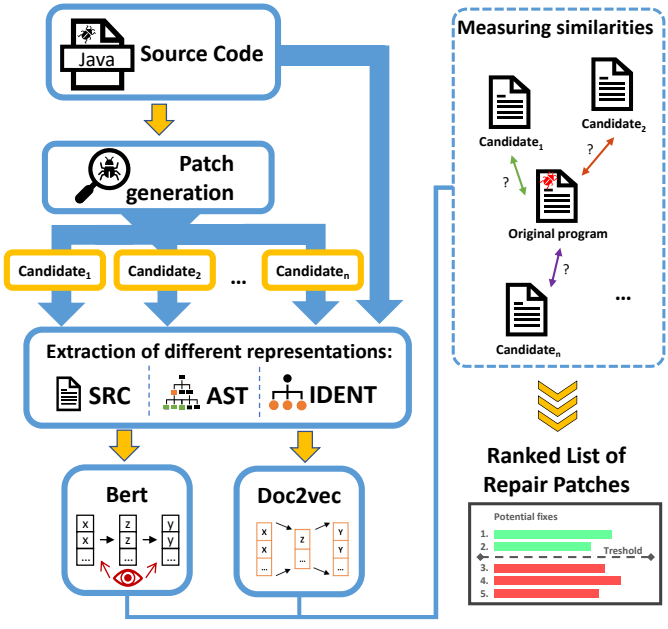

Fig. 2. Illustration of the different embedding approaches

B. Code Representations

For each program in the database, we obtained three different textual representations to measure similarity. We already described these representations in more detail in our previous works [24], [25]. Similar representations are widely used in other research experiments, such as [2], [26]. Here we present a brief summary of these representations:

1) SRC: For a given granularity we consider a source code fragment as a sentence. This sentence is split [27], [28] into bag of words representation along special characters (e.g.: *"("*, *"["*, *"."* etc.) and compound words by the camel case rule.

2) AST: Initially an Abstract Syntax Tree (AST) is constructed on which the node types are printed in a pre-order fashion.

3) IDENT: As before, an AST is constructed, but now for every node we consider its sub-tree and print the *values* of the leaf (terminal) nodes. Thereafter for a given sentence, constant values (literals) are replaced with placeholders, corresponding to their types.

The publicly available JavaParser[1] was used to generate the representations. It is a lightweight tool made for the analysis of Java code. The AST and IDENT representations were generated separately using only the source code of the examined projects.

C. Learning Document Embeddings

First, from every program its representations (SRC, AST and IDENT) is extracted. The experiments were conducted on class level granularity (in case of Java it's usually the same as file level), so the original program and the fixed program is compared in class level. Consider the original (buggy) programs and the potentially fixed programs as documents on which the measurements are conducted. Document embeddings (N dimensional vectors) are learned for each representation, on which similarities are computed. These embeddings are produced with two different techniques: Doc2vec and Bert.

Doc2vec: is a fully connected neural network, which uses a single hidden layer to learn document embeddings. We feed the input documents to this neural network for each representation and it computes the embedding vectors, on which conceptual similarity can be measured. On the obtained embeddings (vectors containing real numbers) similarity is measured with the *COS3MUL* metric, proposed in [29]. According to the authors positive words still contribute positively towards the similarity, negative words negatively, but with less susceptibility to one large distance dominating the calculation. The metric is applicable to document level, because the Doc2vec model is very similar to Word2vec: instead of using just surrounding words to predict the next word, it also adds another feature vector, which is unique for every sentence. This way a single word can have different embeddings in different sentences.

Bert: is a language representation model which – unlike Doc2vec – is designed to train a deep, bidirectional neural network. It is designed in a way, so that it is able to capture the meaning of a word based on its context. Moreover, it can also identify different meanings of the same word. For our experiment we used a pre-trained BERT model [2], which was trained on a large corpus (Wikipedia + BookCorpus) by researchers at Google. In our experiment we used the bert-as-service API [3], mainly due to its easy installation and usage. After starting the service, only a few lines of code is enough the get the encoded representation of a given source code. The service itself consists of two distinct parts, the server and the client. To start the server one should specify its parameters, among which there is one called *max_seq_len*. According to the documentation, it sets *the maximum length of a sequence*, meaning longer sequences will be trimmed on the right side. We experimented with this value, and found that if it is too low, the result is an over-trimmed program, contrary if it is too high, the program will be padded. With this in mind, we chose max_seq_len to the average length of the input text. For the SRC, AST and IDENT representations it was 360, 140, 160 respectively.

IV. Results and Discussion

In this section, we evaluate and discuss the featured text-based models and source code representations. First we present our patch recommendation system. Next, to further examine the featured techniques, we used the obtained embeddings to create a classifier for the patches.

A. Patch Recommendation

Table I shows the ranked lists of patches obtained using different source code representations. Due to space limitations and to stick to the results that seem more important we do not display every bug in the database, the reader can study these in the original publication [23]. Thus we picked three bugs and examined the four most similar generated patches for these, but want to highlight that this choice is arbitrary. In the table headers one can find the signatures of the code representations and the short name of the bugs. The rows correspond to the ordering - the element in the first cell is the most similar to the original program, while the element in the last cell is the most varied. The values in the cells indicate the name of the repair tool and the p_N notation denotes the N-th generated patch of this tool.

Let us compare the two tables. The first obvious difference is, that in case of Bert the cells in case of the AST representation are empty. This is due to the fact, that in these cases we couldn't define an ordering, because every similarity value was practically 1.0. High similarity values were typical throughout the experiment, but the difference between them was distinguishable. Therefore, we cannot really compare the two techniques in case of AST representation. Although the

[1]https://github.com/javaparser/javaparser

[2]https://storage.googleapis.com/bert_models/2018_10_18/cased_L-12_H-768_A-12.zip

[3]https://github.com/hanxiao/bert-as-service

TABLE I
RESULTS FEATURING THE CORPUS BUILT FROM DIFFERENT REPRESENTATIONS OF THE SOURCE CODE, SIMILARITIES OBTAINED WITH **DOC2VEC** AND **BERT**

DOC2VEC

	SRC			AST			IDENT		
	quicksort	*lis*	*shortest_p*	*quicksort*	*lis*	*shortest_p*	*quicksort*	*lis*	*shortest_p*
1	arja p_5	rsrepair p_5	jgenprog p_1	mutation p_1	rsrepair p_5	jgenprog p_1	mutation p_1	rsrepair p_5	jgenprog p_1
2	rsrepair p_1	statement p_1	arja p_4	arja p_1	statement p_1	arja p_4	tibra p_2	statement p_1	arja p_4
3	mutation p_1	tibra p_1	arja p_5	nopol p_1	tibra p_1	arja p_2	arja p_5	tibra p_1	arja p_2
4	arja p_1	arja p_2	arja p_2	arja p_3	arja p_2	arja p_5	rsrepair p_1	arja p_2	rsrepair p_1
5					...				

BERT

1	rsrepair p_1	rsrepair p_5	jgenprog p_1	-	-	-	mutation p_1	tibra p_1	jgenprog p_1
2	arja p_5	tibra p_1	arja p_4	-	-	-	tibra p_1	statement p_1	rsrepair p_4
3	arja p_1	statement p_1	arja p_5	-	-	-	arja p_5	rsrepair p_5	arja p_4
4	jkali p_1	rsrepair p_2	arja p_2	-	-	-	rsrepair p_1	rsrepair p_2	rsrepair p_1
5					...				

quantitative evaluation of these similarity lists is not self-evident, we used the well-known Jaccard similarity [30], also known as Intersection over Union, to quantify the results. We examined the top_5 items of each list, treated these as sets and measured Jaccard similarity on them. Afterwards these metric values were averaged and gave the cumulative similarity value of 0.72. Since the maximum value of the Jaccard similarity is 1.0, this value is quite high. If we consider the results obtained from the IDENT representation of the source code, it can be seen, that in case of problems *quicksort* and *shortest_p* the overlap between the similarity lists is considerable. However the lists of *lis* seems to be different for the first sight, after further investigation we can come to that conclusion, that they differ only in one element. However the IDENT representation in some cases behaves differently, the SRC results are very much alike. For example in case of the bug called *lis*, both Doc2vec and Bert placed the *rsrepair_p_5* patch to the first place of the similarity list. Also note, that if we would switch the placement of the second and third cells in any of the two tables (for this bug), we would get the similarity list of the other table. Based on these observations, we can answer our first research question.

Answer to RQ1: According to the data shown in Table I, Doc2Vec and Bert produces similar results, thus cannot say that one is better than the other. We note that the comparison is not fair, since Bert is a pre-trained model trained on natural language text, while Doc2vec is trained on source code. The simplicity of patches may be another reason why the complexity of the Bert model did not pay off. We expect that in the future, where APR methods can produce complex patches instead of one-liners, the language understanding nature of Bert may be advantageous.

Due to the small number of bugs in the database, these results cannot be generalized in every case. However we would like to emphasize, that after further examinations (for every bug, not just the three shown) of the two similarity lists we didn't find a *significant* difference. Comparing document embeddings is unclear to this day, also giving systematic approach to compare the performance of the two techniques isn't possible in case of ranking, only by manual inspection - that we had no opportunity to do so. Because our observations are based on very limited data, in the future we would like to repeat the experiments with a larger database with more patches. Another topic of research might be to fine-tune the Bert model on source code and not on natural language corpus. However variable namings in Java are usually expressive, teaching on source code have benefits, such as domain specific vocabulary or the interpretation of special programming characters [31].

In this experiment we decided to continue with the Doc2vec embedding method. The major decisive factor was, that using it, we could get meaningful results even for the AST representation. Also note, that we couldn't quantify the usefulness of the obtained similarity lists. This is due to the fact, that the current state-of-the art APR tools repair bugs correctly with similar behavior. In our case this means that for a bug the repair tools either created *only* correct patches or incorrect ones. We did not see such a bug, where both correct and incorrect patches were present. For this reason, we decided to use the obtained similarity list as a classifier, where evaluation is given.

B. Classification of Correct Patches

To use the similarity list as a classifier - which decides whether a patch is correct or not - the only modification has to be done is to define a threshold, and we consider a program to be correct if it is above this number, otherwise as incorrect. To evaluate our method, we used the well-known metrics of statistics: *Negative Predictive Value* (Equation 1), *Positive Predictive Value* or *precision* (Equation 2), *recall* (Equation 3) and *F1 metric* (Equation 4). In the formulas True Positive (TP) and True Negative (TN) values can be interpreted as the following: the former when the algorithm tells from a correct

program that it is correct, while the latter when it tells from an incorrect one that it is faulty. Obviously a good classifier has high TP and TN values, but notice that maximizing the TN value is more interesting, because it means that an incorrect program has been filtered out. The FP value in our case means that the algorithm classifies a program as a correct one, but in reality it is incorrect. This behavior is not very significant, since the original APR tool also classified this output as a correct one - so for this particular patch we haven't been able to make a good decision. However, we are not so lenient about FN, which means that our algorithm tells from a correct program that it is incorrect. This is obviously a serious error, as an APR tool has been working on generating this patch, which will be *mistakenly* filtered out.

$$NPV = \frac{TN}{TN + FN} \quad (1) \qquad PPV = \frac{TP}{TP + FP} \quad (2)$$

$$recall = \frac{TP}{TP + FN} \quad (3) \qquad F_1 = \frac{PPV * recall}{PPV + recall} \quad (4)$$

We would like to emphasize, that the goal of this experiment is to examine whether threshold T exists, which classifies the patches. However finding this threshold automatically might be a difficult task and its solution is out of the scope of this paper. The T value in this work is obtained from empirical observations only, we did not searched for a method, which defines the threshold.

The obtained results for Doc2vec can be found in Figure 3. The upper left corner of the squares corresponds to TP, the lower left to TN, the upper right to FP and the lower left to FN. One can observe that the FN value is the lowest in the case of SRC representation, thus it can be considered as the most effective representation (also note, that the value of the F1 metric reached its peak here).

Answer to RQ2: The SRC representation seems to be prevalent in correct patch identification, while others proved to be much less effective.

One can argue, that this is due to the fact, that fixes are often limited to a single line, and in some cases only a single character is affected (eg. > instead of < in an *if* structure). Such small modifications cannot be detected by the IDENT or AST representations, since they work at a higher level of abstraction. This is especially true for AST, since it only works with types of the abstract syntax tree. Although the IDENT representation works on identifiers of the tree, this does not include such fine modifications (e.g. in the former case, the smaller symbol will be an internal vertex of the abstract syntax tree, so it won't appear at all in the IDENT representation).

Interpreting the results, we found that 16 of the wrong patches have been filtered out. A total of 62 patches were generated for the 14 bugs, of which 27 were correct (since an APR device can generate multiple patches, and several such devices were executed for the same buggy program).This means that 32 incorrect patches have been generated, of which exactly half have been detected correctly.

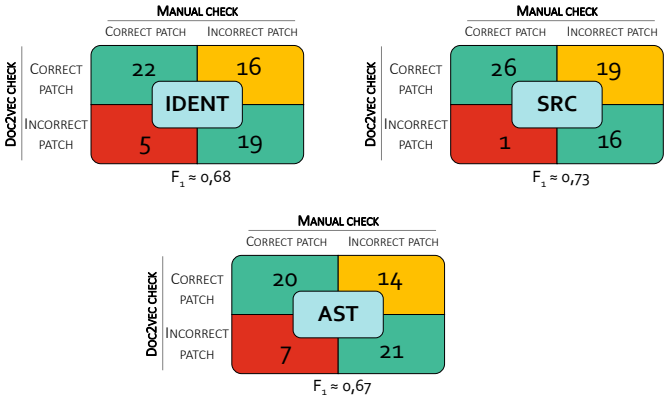

Fig. 3. Evaluative metric values obtained from the similarities measured with the Doc2vec model for the three representations

Answer to RQ3: Based on our data we conclude that similarity-based patch validation can filter out incorrect patches.

The NPV metric value is 0.94, so for an incorrect patch, the algorithm is so likely to say that it is indeed incorrect. The PPV value is 0.57, which means that the algorithm classifies about every second program correctly, if it belongs to the class of correct patches.

It is clear from the figure, that a single patch has been misclassified. Let us examine this case in greater detail, as the evaluation database is relatively small. One can observe this patch on Figure 4. The patch was generated by the Cardumen tool for a bug called MergeSort. The manual patch replaced the original `== 0` with `<= 1`. After a brief consideration one can conclude, that the resulting fix is correct, but less readable (to humans). This is a recurring problem in G&A approaches: is it important that the generated patch be readable to humans when it is working [15], [32]? As many of the potentially improved programs still prove to be defective, human review is inevitable. Thus, readability seems to be an important aspect. Based on these, it is up to the Reader to decide whether labeling the patch below as incorrect was in fact a mistake.

```
@@ -20,11 +20,17 @@

public static ArrayList<Integer> mergesort(ArrayList<Integer> arr) {

    - if ((arr.size()) == 0) {
    + if (((arr.size()) / 2) == 0) {
        return arr;
    } else {
        int middle = arr.size() / 2;

        // ...

        return merge(left, right);
    }
}
```

Fig. 4. A repair patch generated by Cardumen

V. Related Work

To generate repair patches as simple as possible, has already mentioned in many works [11], [15], [33]. This makes the repaired programs more understandable to humans. Such codes that are generated by APR tools without any effort to make them readable are called "alien code" [34]. Although, their subsequent maintenance may be costly, according to a recent study [12] 25.4% (45/177) of the correct patches generated by APR techniques are syntactically different from developer-provided ones.

In previous works [9], it has pointed out that overfitting in the APR domain often occurs. It is also known that there are bugs that do not occur under "lab conditions" [35]. Test cases are very important in the APR domain, since in some cases even patching is based on these [9], [22]. Other approaches also exists, which generate patches by learning human-written program codes [36], [37]. While such approaches have shown promising results, they have recently been the subject of several criticisms [32].

Automatic program repair is in its infancy, with many challenges [38], maybe one of the most important is deciding the correctness of a candidate patch [18], [19]. In a recent study [20] authors highlighted this issue as an open question. To the best of our knowledge, we are the first who employed embedding methods to tackle with this problem.

Recommendation systems are not new to software engineering [39]–[41], presenting a prioritized list of most likely solutions seems to be a more resilient approach even in traceability research [24], [25], [42].

During the recent years of natural language processing, word2vec [43] has become a very popular approach of calculating semantic similarities between various textual holders of information [2], [16], [26], [44]–[48]. In these works, initially word embeddings are obtained and then the authors propagate these to larger text body (e.g.: sentences, documents). Although Doc2vec [49] not enjoys the immense popularity of word2vec, it is still well-known to the scientific community [24], [25], [50]–[53], although they are much less prevalent in the field of software engineering. Similarly to Doc2vec, the use of Bert [17] in this specific domain is not very widespread.

Related research also focuses on artifacts written in natural language and use NLP techniques for various purposes and some even employ word embeddings. In [45], the authors propose an architecture where word embeddings are trained and aggregated in order to estimate semantic similarities between documents which they used for bug localization purposes. Document embeddings can be used to find similarity between sentences/paragraphs/documents [26]. In the software engineering domain, this can also be useful in clone detection.

So far not many authors have dealt with patch validation in the research community. In a recent study [10] authors assessed reliability of author and automated annotations on patch correctness assessment. They first constructed a gold set of correctness labels for 189 patches through a user study and then compared labels generated by author and automated annotations with this gold set to assess reliability. They found that although independent test suite alone should not be used to evaluate the effectiveness of APR, it can be used to augment author annotation. Although this study shows some resemblance to our paper, the work of Xiong et al. [13] is the closest to ours. Their goal is to reduce the number of incorrect patches generated in the APR domain. Their core idea is to exploit the behavior similarity of test case executions. The passing tests on original and patched programs are likely to behave similarly while the failing tests on original and patched programs are likely to behave differently. Based on these observations, they generate new test inputs to enhance the test suites and use their behavior similarity to determine patch correctness. With this approach they successfully 56.3% of the incorrect patches to be generated. Our current approach differs from these in many aspects. First, our approach does not prevent patches to be generated, because it works after the patch generation process, though it can be integrated in an APR tool. Next, they measured the similarity of test case executions, while we did not make any assumptions on these, worked purely on the source code. Also our approach uses document embedding methods, not syntactic or semantic metrics like Cosine similarity or Output coverage [54]. Other Syntactic and semantic distances can also be applied, like in the tool named Qlose [21]. These metrics have several limitations, like maximal lines of code to handle or that they need manual adjustment. On the other hand, the use of document embeddings offers a flexible alternative. Opad [22] (Overfitted PAtch Detection) is another tool, which aims to filter out incorrect patches. However their approach is totally different from ours, we briefly summarize their work. Opad uses fuzz testing to generate new test cases, and employs two test oracles to enhance validity checking of automatically-generated patches. Anti-pattern based correction check is also a possible approach [55].

VI. Conclusions

Natural language processing methods are widely applied in software engineering research, even in the APR domain. In this paper, we employ document/sentence embedding methods on source code to qualify the reliability of candidate patches. Since these methods are intended for natural language texts, we first experimented with various representations of the source code. We found, that plain source code is the most appropriate for this task, probably because it preserves the most information of the original program. Finally, using the obtained similarity lists, we showed that similarity-based techniques can identify incorrect patches. With their help, 45% (16/35) of the incorrect patches were filtered out, thus presenting a valuable alternative for patch-filtering methods. While this study is limited in its data size, we note that today we cope with mostly one liner patches. As APR methods advance, we expect that a more complex language understanding models, like Bert, would be advantageous in deciding patch correctness.

ACKNOWLEDGEMENT

This work was supported in part by the ÚNKP-19-2-SZTE and ÚNKP-19-4-SZTE New National Excellence Programs of the Ministry for Innovation and Technology, by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002), by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 ("Internet of Living Things"). This research was partially supported by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary. László Vidács was also funded by the János Bolyai Scholarship of the Hungarian Academy of Sciences.

REFERENCES

[1] M. M. Matias Martinez, "Writer questions the inevitability of FPs' declining role in inpatient care." *cs.SE*, vol. 29, no. 6, pp. 382–383, oct 1997. [Online]. Available: https://arxiv.org/abs/1410.6651

[2] "Deep learning similarities from different representations of source code," *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, vol. 18, pp. 542–553, 2018.

[3] S. Mahajan, A. Alameer, P. McMinn, and W. G. Halfond, "Automated Repair of Internationalization Presentation Failures in Web Pages Using Style Similarity Clustering and Search-Based Techniques," Tech. Rep., 2018. [Online]. Available: https://www.dmv.ca.gov

[4] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, aug 2017. [Online]. Available: http://link.springer.com/10.1007/s10664-016-9470-4

[5] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix : Fixing Common Programming Errors by Deep Learning," Tech. Rep. Traver, 2016. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/viewFile/14603/13921

[6] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01285008v2

[7] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*. New York, New York, USA: ACM Press, 2015, pp. 24–36. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2771783.2771791

[8] X. B. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007–3033, oct 2018.

[9] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. New York, New York, USA: ACM Press, 2015, pp. 532–543. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2786805.2786825

[10] D. X. B. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On Reliability of Patch Correctness Assessment," in *Proceedings - International Conference on Software Engineering*, vol. 2019-May. IEEE Computer Society, may 2019, pp. 524–535.

[11] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On Learning Meaningful Code Changes Via Neural Machine Translation," *Proceedings - International Conference on Software Engineering*, vol. 2019-May, pp. 25–36, jan 2019. [Online]. Available: http://arxiv.org/abs/1901.09102

[12] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, "How Different Is It between Machine-Generated and Developer-Provided Patches? : An Empirical Study on the Correct Patches Generated by Automated Program Repair Techniques," in *International Symposium on Empirical Software Engineering and Measurement*, vol. 2019-Septe. IEEE Computer Society, sep 2019.

[13] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, may 2018, pp. 789–799.

[14] W. Scacchi, J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani, "Understanding free/open source software development processes," *Software Process Improvement and Practice*, vol. 11, no. 2, pp. 95–105, mar 2006.

[15] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for Simple Program Repairs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, may 2015, pp. 448–458. [Online]. Available: http://ieeexplore.ieee.org/document/7194596/

[16] Q. V. Le and T. Mikolov, "Distributed Representations of Sentences and Documents," Tech. Rep., 2014.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," oct 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[18] F. Y. Assiri and J. M. Bieman, "Fault localization for automated program repair: effectiveness, performance, repair correctness," *Software Quality Journal*, vol. 25, no. 1, pp. 171–199, mar 2017. [Online]. Available: http://link.springer.com/10.1007/s11219-016-9312-z

[19] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, pp. 1219–1219, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3180155.3182526

[20] M. L. Gazzola Luca, Micucci Daniela, "Automatic Software Repair: A Survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, jan 2019.

[21] L. D'Antoni, R. Samanta, and R. Singh, "QLOSE: Program repair with quantitative objectives," Tech. Rep., 2016.

[22] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pp. 831–841, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3106237.3106274

[23] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark," *IBF 2019 - 2019 IEEE 1st International Workshop on Intelligent Bug Fixing*, pp. 1–10, may 2019. [Online]. Available: http://arxiv.org/abs/1805.03454http://dx.doi.org/10.1109/IBF.2019.8665475

[24] V. Csuvik, A. Kicsi, and L. Vidacs, "Source code level word embeddings in aiding semantic test-to-code traceability," in *Proceedings - 2019 IEEE/ACM 10th International Workshop on Software and Systems Traceability, SST 2019*. Institute of Electrical and Electronics Engineers (IEEE), sep 2019, pp. 29–36.

[25] V. Csuvik, A. Kicsi, and L. Vidács, "Evaluation of Textual Similarity Techniques in Code Level Traceability," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11622 LNCS. Springer Verlag, 2019, pp. 529–543.

[26] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pp. 87–98, 2016.

[27] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1754–1780, dec 2014.

[28] "Can better identifier splitting techniques help feature location?" in *IEEE International Conference on Program Comprehension*. IEEE, jun 2011, pp. 11–20.

[29] "Linguistic Regularities in Sparse and Explicit Word Representations," Tech. Rep., 2014.

[30] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, *Using of jaccard coefficient for keywords similarity*, 2013, vol. 2202. [Online]. Available: https://www.researchgate.net/publication/317248581

[31] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained Contextual Embedding of Source Code," 2019. [Online]. Available: http://arxiv.org/abs/2001.00059

[32] M. Monperrus and Martin, "A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. New York, New York, USA: ACM Press, 2014, pp. 234–242. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2568225.2568324

[33] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities," Tech. Rep., 2017. [Online]. Available: http://arxiv.org/abs/1707.04742

[34] M. Monperrus, "Automatic Software Repair: a Bibliography," vol. 51, pp. 1–24, 2018. [Online]. Available: http://arxiv.org/abs/1807.00515{\%}0Ahttp://dx.doi.org/10.1145/3105906

[35] P. Hooimeijer and W. Weimer, *Modeling bug report quality*. New York, New York, USA: ACM Press, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1321631.1321639

[36] X. B. D. Le, D. Lo, and C. L. Goues, "History Driven Program Repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, mar 2016, pp. 213–224. [Online]. Available: http://ieeexplore.ieee.org/document/7476644/

[37] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings - International Conference on Software Engineering*. IEEE, may 2013, pp. 802–811. [Online]. Available: http://ieeexplore.ieee.org/document/6606626/

[38] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, sep 2013. [Online]. Available: http://link.springer.com/10.1007/s11219-013-9208-0

[39] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. New York, New York, USA: ACM Press, 2016, pp. 165–176.

[40] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, jul 2010.

[41] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

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

[43] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *NIPS'13 Proceedings of the 26th International Conference on Neural Information Processing Systems*, vol. 2, pp. 3111–3119, dec 2013.

[44] N. Mathieu and A. Hamou-Lhadj, "Word embeddings for the software

[53] R. A. DeFronzo, A. Lewin, S. Patel, D. Liu, R. Kaste, H. J. Woerle, and U. C. Broedl, "Combination of empagliflozin and linagliptin as second-engineering domain," *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, pp. 38–41, 2018.

[45] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 404–415.

[46] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically Enhanced Software Traceability Using Deep Learning Techniques," in *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*. IEEE, may 2017, pp. 3–14.

[47] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports," in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*. IEEE, oct 2016, pp. 127–137.

[48] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*. IEEE, may 2017, pp. 438–449.

[49] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," Tech. Rep., 2013.

[50] Z. Zhu and J. Hu, "Context Aware Document Embedding," jul 2017.

[51] A. M. Dai, C. Olah, and Q. V. Le, "Document Embedding with Paragraph Vectors," jul 2015.

[52] S. Wang, J. Tang, C. Aggarwal, and H. Liu, "Linked Document Embedding for Classification," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management - CIKM '16*. New York, New York, USA: ACM Press, 2016, pp. 115–124.
line therapy in subjects with type 2 diabetes inadequately controlled on metformin," *Diabetes Care*, vol. 38, no. 3, pp. 384–393, jul 2015.

[54] X. B. D. Le, D. H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. Part F130154, pp. 593–604, 2017. [Online]. Available: https://doi.org/10.1145/3106237.3106309

[55] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. New York, New York, USA: ACM Press, 2016, pp. 727–738. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2950290.2950295