Exploring Plausible Patches Using Source Code Embeddings in JavaScript

Viktor Csuvik, Dániel Horváth, Márk Lajkó, László Vidács

MTA-SZTE Research Group on Artificial Intelligence
University of Szeged, Szeged, Hungary

{csuvikv,hoda,mlajko,lac}@inf.u-szeged.hu

Abstract—Despite the immense popularity of the Automated Program Repair (APR) field, the question of patch validation is still open. Most of the present-day approaches follow the so-called Generate-and-Validate approach, where first a candidate solution is being generated and after validated against an oracle. The latter, however, might not give a reliable result, because of the imperfections in such oracles; one of which is usually the test suite. Although (re-) running the test suite is right under one's nose, in real life applications the problem of over- and underfitting often occurs, resulting in inadequate patches. Efforts that have been made to tackle with this problem include patch filtering, test suite expansion, careful patch producing and many more. Most approaches to date use post-filtering relying either on test execution traces or make use of some similarity concept measured on the generated patches. Our goal is to investigate the nature of these similarity-based approaches. To do so, we trained a Doc2Vec model on an open-source JavaScript project and generated 465 patches for 10 bugs in it. These plausible patches alongside with the developer fix are then ranked based on their similarity to the original program. We analyzed these similarity lists and found that plain document embeddings may lead to misclassification - it fails to capture nuanced code semantics. Nevertheless, in some cases it also provided useful information, thus helping to better understand the area of Automated Program Repair.

Index Terms—Automatic Program Repair, Patch Correctness, Code Embeddings, Doc2vec, Machine learning

Fig. 1: A high level overview of the proposed approach

I. INTRODUCTION

Automated Program Repair (APR) in the research direction has recently reached new heights with promising results recorded [1], [2]. Nothing shows this better than the many program repair tools that have been implemented and the papers that has been published [3]–[8]. Many of these tools follow the Generate-and-Validate (G&V) approach, which first localizes the exact location of the bug in the source code, typically with the help of the test suite, creating a list of the most *suspicious* parts of the program to repair. The intuition is that if the suspicious parts are repaired, the program will work correctly. After localization patch candidates are generated usually by search-based methods. To validate the generated patches, an oracle is needed which can reliably determine if the repair is really correct and can detect if over- or underfitting has occurred. Test suites are widely used as affordable approximations of this oracle in the APR field, therefore (re-)running the tests validates whether the repair
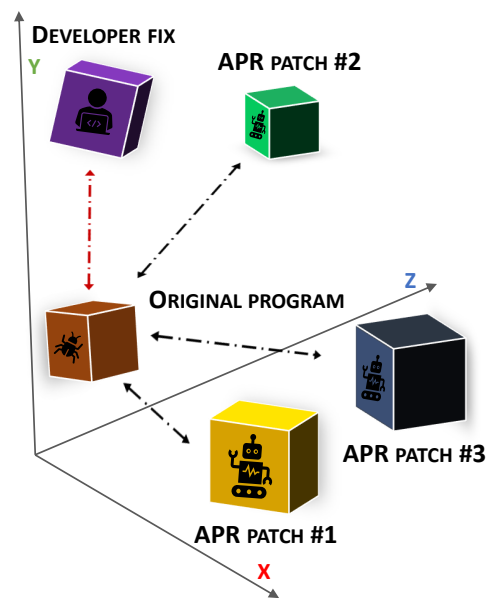
process was successful or not. A program is marked as a plausible patch, if it passes all the available test cases. This latter condition gives no assurance that the program is *correct*, since over- and underfitting [9]–[11] often occurs, resulting in inadequate patches.

Recent works [12]–[17] on patch correctness assessment show the importance of it. Most of these works utilize a similarity concept (i.e. Doc2Vec [18], Bert [19], code2vec [20]) and compute the similarity of software artifacts (e.g. test case execution traces or generated patches). Deciding the correctness of a candidate patch [21], [22] is one of the current challenges in automated program repair, and today considered as an open question [23]. However some repair tools try to generate only real correct patches using carefully-designed (e.g., fine-grained fix ingredients) repair operators or test-suite augmentation, but none of these approaches are able to eliminate overfitting completely [24], [25].

In this work instead of providing another technique for testing the correctness of patches, we are looking at whether

similarity is an appropriate approach to tackle with the patch correctness problem. To do so, we examine not just the generated plausible patches, but the developer fix as well. In Figure 1 one can observe a similarity-based approach, where plausible patches are compared to the original program. Using source code embeddings allow us to place these textual documents into some high-dimensional space, where usual similarity measures can be applied. For example in Figure 1 one can observe, that APR patch #2 lies closer to the developer patch than the other patches, thus it might be better in some way. Of course in higher dimensions, the similarity metric should be well defined, and the exact meaning of each dimension can not be interpreted in most cases. In this work we measured similarity between the generated patches and the original program and also included the developer fix in the comparison process, thus seeking for answers to the following research questions:

RQ1: How effectively can the similarity-based approach be used to validate plausible patches?

RQ2: Do correct patches always show great similarity with the original program?

Our main motivation is to create an exploratory analysis about the similarity-based patch validation. The assumption of similarity-based techniques is 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 single line code modifications, thus leaving most of the original source code intact. When repairing a program, it is preferable to construct patches which are simple and readable that is, to be more similar to the original code base [26]. A well experienced software developer would share the same goal: fix the issue in a way and style that it integrates with the original code base. This is because responsible software maintainers would review and inspect a patch carefully before accepting it [27] – which occurs only if they judge that the patch is correct and safe [26]. Although textual and structural similarity does not imply that the constructed patch is *simple* for human software developers, Csuvik et al. [13] found that in some way, similarity indicates understandability and if a patch is more understandable, its chance of being *correct* is higher. Their approach used the same ranking method as ours, except our experiments are conducted on JavaScript. The contributions of the paper include (1) quantitative and qualitative analysis of plausible patches for JavaScript (2) adaptation of a similarity-based patch filtering method (3) manual annotation of 465 patches. Although we do not present a new technique that can be used for patch filtering, we are adapting one to JavaScript and analyzing its usability in depth.

In this paper we used bugs from the BugsJS dataset [28] which contains 453 reproducible JavaScript bugs from 10 open-source Github projects. The tool we used is an adaptation of the original GenProg [29] method supplemented with JavaScript-specific repair operators[1]. We used the generated

---

[1]The complete description of the tool is beyond the scope of the current paper, the interested reader may find more information about it in https://github.com/GenProgJS.

plausible patches for 10 distinct bugs all from the Eslint project. We restricted our experiments to the Eslint project because it is the largest project in the BugsJS dataset, it contains the most single-line errors.

The paper is organized as follows. In the next section we present the experiment setup including the applied method (Section II-A), the used embedding technique (Section II-B) and the sample plausible patches (Section II-C). Evaluation and analysis are presented in Section III. Related work is discussed in Section IV, and we conclude the paper in the last section.

II. EXPERIMENTAL SETUP

In this section we describe the used approach to determine the usefulness of similarity based patch validation. A high-level overview of the proposed process can be found in Figure 2. First an APR tool creates plausible patches, usually more than one. In our case the tool always ran for 30 generations resulting in a high number of plausible patches for each bug. From the original program and from the generated potentially fixed programs, we extract the faulty line and a small environment of it - this snippet of code will serve as a basis for calculating similarity. After a Doc2Vec model is trained, for every code snippet an N dimensional vector is created on which one can measure similarity. These vectors contain information about the meaning, environment, and context of a word or document. The generated plausible patches then lined up alongside with the developer fix, based on the similarities calculated previously. Based on this list we can analyze which version of the fixed program is the most similar to the original one - the one created by an APR tool or a developer fix.

In the next subsections the evaluation procedure is presented first, then the method used to measure similarity, and finally the used plausible patches are described.

A. Method

For the investigated bugs, determining, whether a fix for it is correct or not, was usually quite easy. In general to answer this question again can be a challenging, but in our case the generated patches were one-liners and usually it was clear whether the APR tool generated an overfitted patch or a correct one. Although determining which fix is better than the other one can be tricky. A rule-of-thumb can be that the simpler a patch the better. This, of course, supports similarity-based validation, but it may not be the right solution in all cases. Ranking plausible patches alongside with the developer patch may also point out that a human written patch differs greatly from the original source code. This does not necessarily mean that the developer made a mistake: it might be that he adapts a new approach that was never used before in the code base, or simply made a refactoring of considerable size.

Our main question is, whether it is true — from the perspective of Doc2Vec — that the developer fix lies close to the original program. Current state-of-the-art APR applications still fail to repair real complex issues, thus the demand for

simple patches may be desirable. To measure the quality of the ranking, we used the Normalized Discounted Cumulative Gain (*nDCG*) metric [30], which is computed as:

$$nDCG_p = \frac{DCG_p}{IDCG_p} \quad (1)$$

Where *DCG* stands for Discounted Cumulative Gain, *IDCG* stands for Ideal *DCG* and *p* is a particular rank position. *DCG* measures the usefulness, or gain, of a document based on its position in the result list. *IDCG* basically is the maximum possible *DCG* value that can be achieved on a ranked list - this is done by sorting all relevant documents in the corpus by their relative relevance. Since the similarity lists vary in length (the number of plausible patches is different for each bug), consistently comparing their performance with *DCG* is not possible. To be able to compare these lists, cumulative gain at each position for a chosen value of *p* should be normalized, thus resulting in the *nDCG* metric defined above in Equation 1. The definition of *DCG* and *IDCG* is presented in Equation 2 and Equation 3 respectively.

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i}-1}{log_2(i+1)} \quad IDCG_p = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i}-1}{log_2(i+1)}$$
$$(2) \qquad\qquad\qquad (3)$$

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

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

In addition to these, intermediate categories are also conceivable: e.g. -0.5 would mean that a patch is probably incorrect, but we were not sure about that, because the lack of domain knowledge about the system examined. Two experienced software developers separately annotated the generated patches, they did not have the chance to influence each other. In cases where individual scores differed a third expert decided on the correctness of the patch. These annotated relevance scores are available in the online appendix of this paper[2].

B. Learning Document Embeddings

For every bug a small environment of the faulty line was selected, and this was embedded using Doc2Vec. This code environment includes the faulty line itself and three lines in front of and behind it. For our training data we tokenized this small code fragment with a simple regular expression, which separated words and punctuations, except for words with the dot '.' (member) operator. For a simple code example like: *function foo () { return this.bar; }* the tokenized version would be: *'function', 'foo', '(', ')', '{', 'return', 'this.bar', ';', '}'* . Doc2Vec is a fully connected neural network that uses a single hidden layer to learn document embeddings (N dimensional vectors). We feed the input documents to this neural network, and it computes the embedding vectors, on which conceptual similarity can be measured. For our Doc2Vec configuration we used a vector size of 256, which basically is the dimensionality of the feature vectors. Window size of 5 was used, which tells Doc2Vec the maximum distance between the current and predicted word. Every word with a frequency of less than 2 was ignored. As for the training, the model was trained for 50 epochs. Every other parameter was left as default. On the obtained embeddings (vectors containing real numbers) similarity is measured with the *COS3MUL* metric, proposed in [31]. 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.

C. Sample plausible patches

The BugsJS dataset [28] contains 453 reproducible JavaScript bugs from 10 open-source Github projects. The dataset contains multi-line bugs as well, which are beyond the scope of the current research. There are 130 single-line bugs, but not every one of them are "repairable", because of
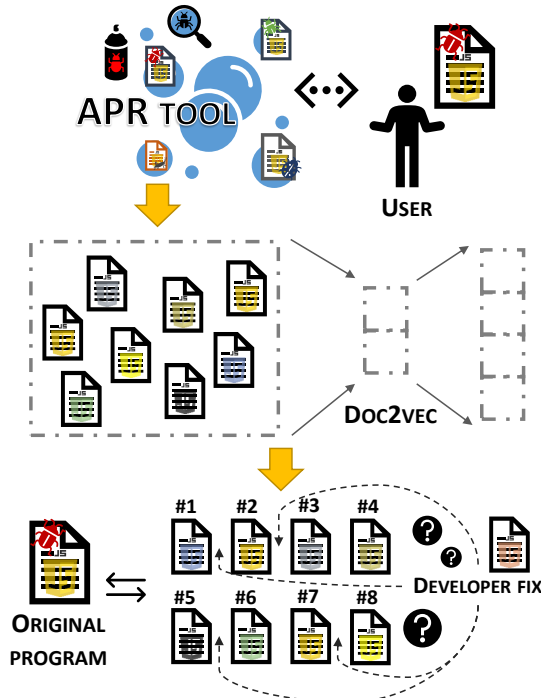

Fig. 2: Illustration of the implemented process

the lack of failed test cases. Thus, the number of single-line bugs, for which there is at least 1 failed test is 126 (and 94 only comes from the Eslint project). BugsJS features a rich interface for accessing the faulty and fixed versions of the programs and executing the corresponding test cases. These features proved to be rather useful for the comparison of automatically generated patches with the ones which were created by developers. We limited our experiments strictly to the Eslint project because it is the largest project in the BugsJS dataset, it contains the most single-line errors. The automatic repair tool which we used was able to repair 10 bugs from 94 in the Eslint project. Since the tool was configured to run for 30 generations in every case (so it does not stop at first when a fix is found), there was a high number of repair candidates in most cases of the runs as can be seen in Table I. In the first column one can find the id of each bug and next to it how many plausible patches were generated to it. The two remaining columns show the original source code and a fix for it created by a developer.

Each of the examined bugs and their corresponding fixes are one-liners, meaning that the modification that a developer (or APR tool) has to make to fix the bug only affects one line. We can see that there are bugs of different difficulty in Table I: from quite simple where a number had to be replaced (Eslint 47), to quite complex where a conditional expression needed to be supplemented (Eslint 100). The number of generated candidates also varies greatly, this is due to the difficulty of the fixes and the random factor in the GenProg algorithm. In total 465 plausible patches were generated. We checked these patches and found that only three of these are syntactically identical to the developer fix (Eslint 47, Eslint 323 and Eslint 72), although many of them are semantically identical.

It is apparent that for Eslint 221 and Eslint 323 the number of plausible patches is orders of magnitude more than for any other. To explain it let us examine the nature of these bugs. The case of Eslint 221 is quite easy to understand: the return value should be `false`, making it rather simple to generate. We examined the generated patches and found that essentially anything would satisfy this criteria: in JavaScript `0`, `-0`, `null`, `false`, `NaN`, `undefined`, or the empty string (`""`) create an object with an initial value of `false`. On the other hand in case of Eslint 323 the high number of plausible patches is

most probably because of the weak test suite. As we can see from Table I the fix is not quite obvious, but after carefully inspecting the generated fixes we came to the conclusion that every modification on which the `if` condition evaluated to `true` successfully passed testing.

III. RESULTS AND DISCUSSION

In this section we evaluate and analyze the generated patches and similarity lists. First the quantitative evaluation is presented, next we examine some of the patches to get further insights into the nature of the repaired bugs.

A. Quantitative Evaluation

The calculated metric values of *nDCG* described in Equation 1 of Section II-A can be found in Figure 3. The possible values of the metric ranges from 0.0 to 1.0, a higher metric value means better ranking. We can see that in case of Eslint 217 and Eslint 41 the values are 1.0, this is clearly because the developer fix was ranked to the first place in these cases and irrelevant documents were placed on the end of the similarity list. Based on this metric it is clear that in most cases similarity lists hold their place in ranking patches. Let us discuss further the cases where the metric values are lower.

The *nDCG* metric value reach it's lowest point at the Eslint 94 bug. If we take a look at the subplot (e) at Figure 4 we can see that indeed that Doc2vec failed to rank the developer fix at the top of the list. Moreover, most of the patches at the top of the list are incorrect ones, meaning that they hold low relevance. The case of Eslint 100 is different from the previous one. Although it is true that the *nDCG* value of Eslint 94 is 0.67, while for Eslint 100 it is 0.84, compared to others it still seems to be quite low. If we take a look at the rankings at Figure 4 we can see that in this case the developer fix is placed on the second place of the ranked list. So the question arises, what causes the low metric value? The answer is quite obvious: the patch which is placed ahead is an incorrect one, decreasing the metric value drastically. The case of Eslint 221 is also interesting: although the developer patch is placed closer to the end of the list than anywhere else, the *nDCG* metric value is not that low. This is due to the fact that in case of this bug the majority of generated plausible patches are semantically the same as the developer fix, resulting in overall higher relevance scores.

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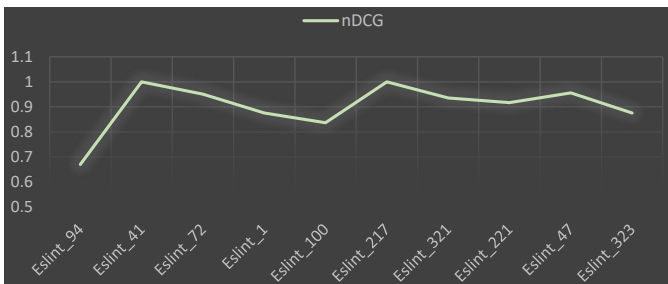

Fig. 3: The nDCG metric value based on each of the examined bugs' ranked list

TABLE I: Plausible patches and their corresponding developer fix in the Eslint project

| Bug Id	#candidate	Original line	Developer fix
Eslint 1	4	`if (name === "Math" || name === "JSON")`	`if (name === "Math" || name === "JSON" || name === "Reflect" )`
Eslint 41	3	`end.column === line.length )`	`( end. line === lineNumber && end. column === line.length ));`
Eslint 47	3	`column: 1`	`column: 0`
Eslint 72	7	`loc: lastItem .loc.end,`	`loc: penultimateToken .loc.end,`
Eslint 94	14	`op.type === "Punctuator" &&`	`( op.type === "Punctuator" || op.type === "Keyword") &&`
Eslint 100	12	`penultimateType === "ObjectExpression"`	`( penultimateType === "ObjectExpression" || penultimateType === "ObjectPattern")`
Eslint 217	4	`if (!options || typeof option === "string")`	`if (!options || typeof options === "string")`
Eslint 221	221	`return parent.static ;`	`return false ;`
Eslint 321	5	`...loc.end.line !== node.loc. end .line &&...`	`...loc.end.line !== node.loc. start .line &&...`
Eslint 323	192	`else if (definition.type === "Parameter" && node.type === "FunctionDeclaration")`	`else if (definition.type === "Parameter")`
Total	**465**		
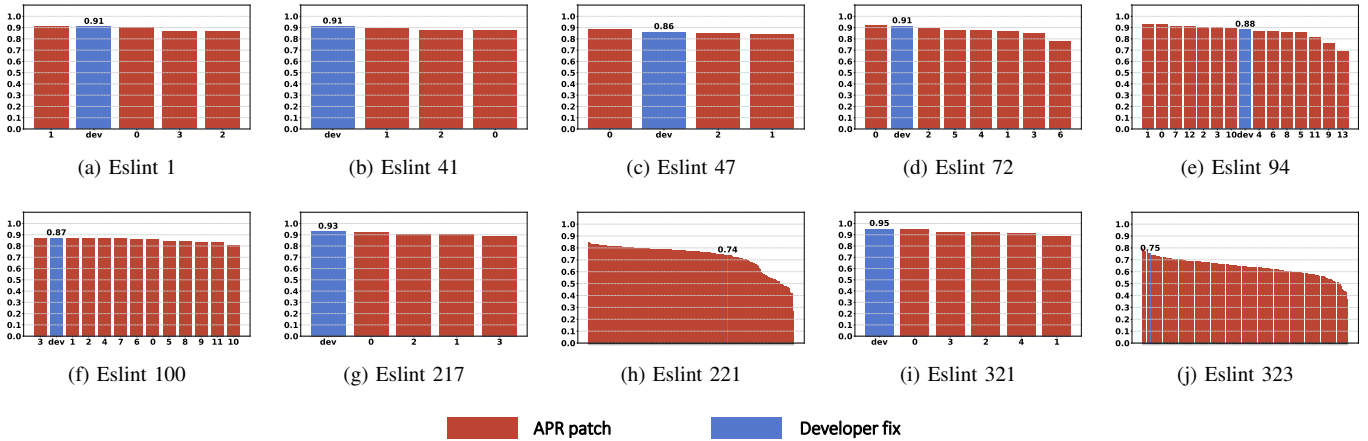
Fig. 4: The developer fix and patches ranked based on their similarity to the original program

(a) Eslint 1 (b) Eslint 41 (c) Eslint 47 (d) Eslint 72 (e) Eslint 94
(f) Eslint 100 (g) Eslint 217 (h) Eslint 221 (i) Eslint 321 (j) Eslint 323

APR patch — Developer fix

B. Qualitative Evaluation

Due to space constraints we cannot analyze manually every patch, however, we will present two of them in this subsection. We would like to emphasize that our goal is not to use the similarity list as a classifier — which decides whether a patch is correct or not — but to analyze the patches. For each of the bugs described below, the developer fix is listed in the Table I. The first patch we are going to examine in more details is the one generated for Eslint 1.

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

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

In Listing 1 one can examine the original line with red background and the line that was generated by an APR tool with green background. In this case the developer fix adds another logical testing in the if condition, allowing the `name` variable to have the value `"Reflect"` as well. The modifications which the APR tool made bears no resemblance to this. At first sight it does show greater similarity with the original line than the developer fix, however the generated code is clearly not the best. First let us understand the generated line of code. The `name` variable contains a string value and it is compared with `<=` relation to another string value. In JavaScript if both values are strings, they are compared based on the values of the Unicode code points they contain. Meaning that every string which begins with a letter in front of S in alphabetical order will evaluate to `true` otherwise `false` - according to this if the `name` variable has the value `"Reflect"` or `"JSON"` it will evaluate to `true`. So far not that bad. Surprisingly changing the logical *or* (`||`) operator to the bitwise or operator (`|`) does not have any effect here, since the bitwise operation `false | true` results in `1` which converted to boolean evaluates to `true`. Similarly true for every case of the logical operation.

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

true that the generated patch is overfitted, it contains valuable information about the repair i.e. gives a hint to the developer which variable might cause the incorrect behaviour.

> **Answer to RQ1:** Based on the observed patches, we would recommend a more sophisticated technique to validate patches than plain source code embeddings, because as we have seen the problem itself is more nuanced and complex than a simple true/false classification.

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). We definitely have to mention that the patches were generated with the use of a single APR tool, it is hard to justify if the conclusions are valid for other tools and multi-line fixes as well. However, defining a threshold and based on this deciding on the correctness of a patch, seems to oversimplify the decision criterion too much. On the other hand, the strive for understandable and simple patches is a reasonable and important aspect of automatic software repair. Generating unreadable patches does not help much with a real-life problem. But if a patch is not too similar to the original program, does it exclusively mean that it is unreadable? On Listing 2 we can observe another code snippet from the Eslint project, but this time we picked the least similar generated patch from the bug Eslint 321. At first glance the two lines seem to be very similar even though in the similarity list it was the last one. The latter does not necessarily mean a big difference, especially if there are very few candidates: in this special case even the last plausible patch shows great similarities with the original program.

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

Listing 2: Original code of Eslint 321 (-), and the least similar automatically generated patch to it (+)

In case of Eslint 321 the developer fix only changed the `end` word to `start`. This is obviously a small bug and is probably due to developer inattention. Though the automatically generated fix also changed this class member it made further changes. First it changed the double quotes (`"`) to single ones (`'`). Next deleted the strict not equal operator (`!==`) to subtraction (`–`). The first change obviously did not affect the meaning of the `if` structure, but neither did the latter, because if the observed two values are equal and we subtract them, the result is 0, which is evaluated to `false` in JavaScript.

> **Answer to RQ2:** The last item in the similarity list can also be a semantically correct one, even though it is less similar to the original program. From this, we can conclude that while similarity-based methods may be suitable for filtering out too many patches, one should use them for classification cautiously bearing in mind the possible misclassification.

In total we examined 465 automatically generated patches as can be seen in Table I. From these 13 were syntactically equivalent to the developer fix and 211 semantically. We found that most of the semantic-matched patches were more similar to the original code than others. This behavior can be observed on Figure 3, where the metric values were calculated using data annotation based on the correctness of each patch. The similarity of the developer fix also tends to be close to the original program as one can see on Figure 4. Our experiments targeted one-line modification and the evaluation was conducted on only one project. These might seem to be limitations, however at the time of writing the paper, there was no available APR tool, which could generate multi-line patches for JavaScript programs. We encourage the interested reader, who is interested in more analysis of the generated patches read the paper of GenProgJS, linked above.

IV. RELATED WORK

To generate repair patches as simple as possible, has already mentioned in many works [15], [26], [32]. According to a recent study [16] 25.4% (45/177) of the correct patches generated by APR techniques are syntactically different from developer provided ones. Other approaches also exists, which generate patches by learning human-written program codes [33], [34]. While such approaches have shown promising results, they have recently been the subject of several criticisms [35].

In automated program repair the evaluation of existing approaches is crucial. Evaluating APR tools based on plausible patches are not accurate, due to the fact of the overfitting issue in test suite-based automatic patch generation. Finding the correct patches among the plausible patches requires additional developer workforce. Liu et al. [36] proposes eight evaluation metrics for fairly assessing the performance of APR tools beside providing a critical review on the existing evaluation of patch generation systems.

In a recent study [37] benchmarks the state of art patch correctness techniques based on the largest patch benchmark so far and gathers the advantages and disadvantages of existing approaches beside pointing out a potential direction by integrating static features with existing methods. Another work where embedding methods were used for ranking candidates is [12], where beside Doc2Vec and Bert, code2vec and CC2Vec were also applied. In this work they investigate the discriminative power of features. They claim that Logistic Regression with BERT embedding scored 0.72% F-Measure and 0.8% AUC on labeled deduplicated dataset of 1,000 patches.

Recommendation systems are not new to software engineering [38]–[40], presenting a prioritized list of most likely solutions seems to be a more resilient approach even in traceability research [41]–[43]. In a recent work [13] candidate patches were ranked based on their similarity and evaluated as a recommendation system. They have proposed a small-scale study on the use of embeddings: leveraging pre-trained natural language sentence embedding models, they claim to have been able to filter out 45% incorrect patches from the QuixBugs dataset.

In a recent study [14] authors assessed reliability of automated annotations on patch correctness assessment. They constructed a gold set of correctness labels for 189 patches through a user study and then compared these labels with automated generated annotations to assess reliability. They found that independent test suite alone might not serve as an effective APR oracle, it can be used to augment author annotation. In the paper of Xiong et al. [17] the core idea is to exploit the behavior similarity of test case executions. The passing tests on original and patched programs are likely to behave similarly while the failing tests on original and patched programs are likely to behave differently. Based on these observations, they generate new test inputs to enhance the test suites and use their behavior similarity to determine patch correctness. With this approach they successfully prevented 56.3% of the incorrect patches to be generated.

Syntactic or semantic metrics like Cosine similarity and Output coverage [44] can also be applied to measure similarity, like in the tool named Qlose [45]. 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 [24] (Overfitted Patch Detection) is another tool, which aims to filter out incorrect patches. Opad uses fuzz testing to generate new test cases and employs two test oracles to enhance validity checking of automatically generated patches. Antipattern based correction check is also a viable approach [46].

V. Conclusions

Patch validation in the APR domain is a less explored area, which holds great potential. Filtering out incorrect patches from the set of plausible programs is an important step forward to boost the confidence towards APR tools. In this paper we experimented with a similarity-based patch filtering approach and conducted a quantitative and qualitative evaluation of it. The similarity between patches was calculated with the use of source code embeddings produced by Doc2Vec. Although the applied approach may be useful when a high number of plausible patches are present, we found that plain source code embeddings fail to capture nuanced code semantics, thus a more sophisticated technique is needed to correctly validate patches. We expect that a more complex language understanding model may be advantageous in deciding whether a patch is correct or not. In future work, we wish to explore whether the use of deep learning techniques constitutes additional advantages.

References

[1] 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

[2] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, nov 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3318162

[3] 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

[4] "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.

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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.

[10] 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

[11] 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

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

[13] V. Csuvik, D. Horváth, F. Horváth, and L. Vidács, "Utilizing Source Code Embeddings to Identify Correct Patches," in *IBF 2020 - Proceedings of the 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing*. Institute of Electrical and Electronics Engineers Inc., feb 2020, pp. 18–25.

[14] 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.

[15] 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

[16] 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.

[17] 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.

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

[19] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, vol. 1, pp. 4171–4186, oct 2019. [Online]. Available: http://arxiv.org/abs/1810.04805

[20] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning Distributed Representations of Code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, jan 2018. [Online]. Available: http://arxiv.org/abs/1803.09473

[21] 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

[22] 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

[23] 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.

[24] 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

[25] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, pp. 1–11, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3180155.3180233

[26] 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/

[27] 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.

[28] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszedes, R. Ferenc, and A. Mesbah, "BugsJS: A benchmark of javascript bugs," in *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, apr 2019, pp. 90–101. [Online]. Available: https://github.com/hakobera/node-v8-benchmark-suite

[29] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. USA: IEEE Computer Society, 2009, p. 364–374.

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

[30] K. Järvelin and J. Kekäläinen, "Ir evaluation methods for retrieving highly relevant documents," vol. 20, 01 2000, pp. 41–48.

[32] 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

[33] 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/

[34] 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/

[35] 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

[36] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, jan 2021.

[37] S. Wang, M. Wen, B. Lin, X. Mao, H. Wu, D. Zou, H. Jin, and Y. Qin, "Automated Patch Correctness Assessment: How Far are We?" *ASE 2020 - Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1166–1178, 2020. [Online]. Available: https://doi.org/10.1145/3324884.3416590

[38] 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.

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

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

[41] 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.

[42] 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.

[43] 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.

[44] 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

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

[46] 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