Predicting Patch Correctness Based on the Similarity of Failing Test Cases

HAOYE TIAN, YINGHUA LI, WEIGUO PIAN, and ABDOUL KADER KABORÉ, University of Luxembourg, Luxembourg
KUI LIU*, Software Engineering Application Technology Lab, Huawei, China
ANDREW HABIB, JACQUES KLEIN, and TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg

How do we know a generated patch is correct? This is a key challenging question that automated program repair (APR) systems struggle to address given the incompleteness of available test suites. Our intuition is that we can triage correct patches by checking whether each generated patch implements code changes (i.e., behaviour) that are relevant to the bug it addresses. Such a bug is commonly specified by a failing test case. Towards predicting patch correctness in APR, we propose a novel yet simple hypothesis on how the link between the patch behaviour and failing test specifications can be drawn: *similar failing test cases should require similar patches*. We then propose BATS, an unsupervised learning-based approach to predict patch correctness by checking patch **B**ehaviour **A**gainst failing **T**est **S**pecification. BATS exploits deep representation learning models for code and patches: for a given failing test case, the yielded embedding is used to compute similarity metrics in the search for historical similar test cases to identify the associated applied patches, which are then used as a proxy for assessing the correctness of the APR-generated patches. Experimentally, we first validate our hypothesis by assessing whether ground-truth developer patches cluster together in the same way that their associated failing test cases are clustered. Then, after collecting a large dataset of 1,278 plausible patches (written by developers or generated by 32 APR tools), we use BATS to predict correct patches: BATS achieves AUC between 0.557 to 0.718 and recall between 0.562 and 0.854 in identifying correct patches. Our approach outperforms state-of-the-art techniques for identifying correct patches without the need for large labeled patch datasets; as is the case with machine learning-based approaches. While BATS is constrained by the availability of similar test cases, we show that it can still be complementary to existing approaches: when combined with a recent approach that relies on supervised learning, BATS improves the overall recall in detecting correct patches. We finally show that BATS is complementary to the state-of-the-art PATCH-SIM dynamic approach for identifying correct patches generated by APR tools.

CCS Concepts: • **Software and its engineering → Software verification and validation**; *Software defect analysis*; Software testing and debugging.

Additional Key Words and Phrases: Program Repair, Patch Correctness, Test Behavior, Patch Semantics

*Corresponding author.

Authors' addresses: Haoye Tian, haoye.tian@uni.lu; Yinghua Li, yinghua.li@uni.lu; Weiguo Pian, weiguo.pian@uni.lu; Abdoul Kader Kaboré, abdoulkader.kabore@uni.lu, University of Luxembourg, Luxembourg; Kui Liu, liukui0811@163.com, Software Engineering Application Technology Lab, Huawei, China; Andrew Habib, andrew.a.habib@gmail.com; Jacques Klein, jacques.klein@uni.lu; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu, University of Luxembourg, Luxembourg.

1049-331X/2022/1-ART1
https://doi.org/10.1145/3511096

ACM Reference Format:
Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kaboré, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. 2022. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2022), 29 pages. https://doi.org/10.1145/3511096

1 INTRODUCTION

Patch overfitting [63, 69] is now acknowledged as one of the major blockers to the adoption of automated program repair (APR) by software practitioners. It refers to the fact that APR-generated patches often overfit to the repair (incomplete) test suite without necessarily generalizing to other test cases. In short, overfitting patches do not implement the desired behavior that the program developers would expect. Consequently, when a generated patch is validated as passing all test cases in the test suite, it is referred to as a ***plausible*** patch. Its correctness, indeed, must still be decided manually by developers. Given that existing APR approaches generate a large number of plausible patches, most of which are actually incorrect, there is a need to develop automated approaches that can filter out incorrect patches or that can rank the correct ones higher to alleviate the burden of manual assessment.

Recently, the literature has proposed various heuristics to predict patch correctness. Csuvik *et al.* [10] translate some empirical observations into a simple assumption for ranking valid patches: correct patches apply fewer changes than incorrect ones. Xiong *et al.* [80] build on the hypothesis that test case dynamic execution behaviours are different between correct and incorrect patches. Other researchers propose to focus on learning, with static features of patches, to filter out incorrect patches. For example, Ye *et al.* [87] proposed such a supervised learning-based approach after investing in careful engineering of patch features. In contrast, Tian *et al.* [72] relied on deep representation learning of code changes for yielding patch embeddings that are fed to a supervised learning system.

Overall, existing research in patch correctness prediction has provided promising performance [75]. Nevertheless, they suffer from various caveats. On the one hand, the state of the art dynamic-based approaches require the expensive execution of test cases, which unfavorably impacts the efficiency of the patch assessment process. Besides, such approaches are challenged in practice by the oracle problem: given a test case, we do not always have an accurate specification of what the output should be [73]. On the other hand, static-based approaches often require a substantial analysis effort to identify adequate features and properties. In addition, machine learning-based approaches require many labeled patch samples (both correct and overfitting patches). They further exhibit issues of generalization beyond the projects they have been trained on [75].

In their seminal study on patch plausibility and correctness, Qi *et al.* [63] have presented Kali, a system that performs "repair" by only removing or skipping code in programs. Kali generated several patches that pass many weak test suites. Our postulate, however, is that Kali-generated patches should be readily-identifiable as *plausible but incorrect* in an APR pipeline: it is unlikely that fixing a program that presents a bug in array iteration would require simply removing whole statements. This calls for research to assess the behaviour of the patch (i.e., what it does) against the nature of the bug (i.e., as expressed by the failing test case specification).

The idea of checking patch behaviour against failing test specification has not yet been fully exploited in the literature. Recent work by Ye *et al.* [87] and Tian *et al.* [72] do not even reason about the test cases. The approach by Xiong *et al.* [80] builds on heuristics that consider the similarity of execution behaviours of passing test cases on original and patched programs. Their work, however, does not try to answer the specific question of ***whether the generated correct patch is actually relevant to the failing test case(s) triggering the repair process***. This is the key hypothesis we introduce and validate:

"When different programs fail to pass similar test cases, it is likely that these programs require similar code changes."

Similarity thus becomes a key challenge: syntactic similarity is not sufficient as it would restrict the search to type-1 or type-2 clones. We have to explore similarity measurements that can capture semantic relationships. Recent work [1, 2, 25, 27, 42] in learning distributed representations of code and code changes have been shown to preserve some semantics (beyond token similarity) and have yielded promising models that were effective on a variety of downstream tasks.

This paper. We build on the aforementioned hypothesis to investigate the possibility of predicting patch correctness by clustering test cases and patches. In this paper, we rely on recent approaches for code representation learning to reason about code and patch similarity.

❶ **[Heuristic]** We propose a novel heuristic on the relationship between patches and their failing test cases. Although the intuition behind this heuristic is basic and hinted at in regression testing literature [4], we are the first to propose and validate it in the APR patch assessment area.

❷ **[Validation]** We present a comprehensive validation of the hypothesis that similar test cases are associated with similar patches. Concretely, we consider the case of developer-written patches in the Defects4J dataset and leverag various distance metrics to perform hierarchical clustering based on the embeddings of test cases and patches.

❸ **[BATS]** Based on the heuristic, we propose BATS (Behaviour Against failing Test Specification), an approach to predict patch correctness by statically checking the similarity of generated patches against past correct patches that correspond to failing test cases which are similar to the failing tests of the bug under resolution. More specifically, given one buggy program with its failing test cases and the APR-generated patches, BATS first enumerates similar failing test cases within the search space of historical bugs. Then, BATS calculates the similarity between the correct patches associated with the identified failing test cases and the APR-generated patches. Finally, APR-generated patches with similarity scores above a predefined threshold t are predicted as correct while patches with similarity scores lower than t are predicted as incorrect. The artifact of this study is publicly available at **https://github.com/HaoyeTianCoder/BATS**.

❹ **[Evaluation]** After collecting a large dataset of plausible patches generated by 32 APR tools or extracted from defects benchmarks, we apply BATS and measure its performance in identifying correct patches.
- Overall, BATS achieves an AUC (Area Under Curve) between ~0.56 and ~0.72 and a recall between ~56% ~84% in identifying correct patches.
- When comparing with a recent supervised learning classifier [72], BATS improves the recall in identifying correct patches by 7 percentage points and achieves an equivalent recall in excluding incorrect patches.
- Comparing against the state-of-the-art dynamic approach PATCH-SIM [80], BATS outperforms it with higher scores of AUC, F1 +Recall and -Recall for the subset of patches where BATS can find similar test cases.
- We note that the performance of BATS is impacted by the search space for finding similar test cases. Therefore, after demonstrating the promise of the proposed heuristic, we show that it is worthwhile to combine BATS with the state-of-the-art approaches in order to improve performance in identifying correct patches. To that end we consider a usage scenario where similar test cases are lacking to apply BATS. Instead, we investigate whether BATS can still be used as a supplement to another approach: when BATS is integrated with

the recent supervised learning classifier [72] , the overall recall in detecting correct patches is improved with 5 percentage points. The experimental results also show that BATS can be complementary to PATCH-SIM [80] to recall more correct and exclude more incorrect patches.

2 MOTIVATION

The promise of APR is to automatically find patches for software bugs without human intervention [20, 22, 58]. This often translates into a generate-and-validate pipeline [7, 21, 30, 39, 41, 46, 47, 76, 77, 81] that consists of two main activities: (i) *Patch generation* which applies various code transformation and search techniques to produce a set of candidate patches, and (ii) *Patch validation* which checks that the candidate patches are relevant to the discovered fault with corresponding test cases. Identifying correct patches among the automatically validated patches, i.e., checking that the functionality of the patched program satisfies the developer's intention, is usually conducted manually. Even recent deep-learning based repair approaches [8, 53, 74, 78] produce patches that require validation, although new research on involving developers early in the process is arising [5, 19].

In a large body of the literature, patch validation is carried out by executing the patched program against the test suite [6, 89]. If a single test case leads to a failure, the patch is discarded as invalid. Rothermel *et al.* [64] have introduced a test case prioritization strategy in order to early-detect invalid candidate patches. In the same vein, Qi *et al.* [62] empirically highlighted that test cases that have been effective in revealing invalid patches should be re-run earlier than others on new patch candidates. Nevertheless, these efforts are directed towards identifying valid (i.e., plausibly correct) patches w.r.t. the test suites. Indeed, a test suite being only a weak approximation of the program specification, patch validation with test suites does not provide guarantees on patch correctness. In practice, the correctness of APR-generated patches is checked manually by the developers. Even in the literature, assessment procedures done by researchers check correctness by using heuristics to compare plausible patches against developer oracle patches [50]. Some works [86] propose to generate more test inputs to strengthen the test inputs and provide more confidence in the generated patches. The oracle problem [28] in test generation however makes the execution results of the augmented test suite an unreliable criterion of patch correctness. Shariffdeen *et al.* [68] thus proposed to reduce the space of patch candidates by simultaneously traversing the spaces of test inputs and patches with path exploration.

3 APPROACH

In this section, we present BATS, our approach to predict patch correctness based on the similarity of the new patch to known correct patches with *similar failing test cases*. Figure 1 gives an overview of the approach. BATS assumes the following inputs: (i) The bug under resolution, its *associated failing test cases*, and *APR-generated plausible patches*, and (ii) Historical bugs with *their failing test cases* and the *associated known correct patches*.

To predict patch correctness, BATS performs the following steps: (1) Pre-process patches and test cases to prepare them for the embedding step, (2) Embed patches and test cases into higher dimensional space, (3) Compute pairwise similarity between *the failing test cases of the bug under resolution* and the *historical test cases*, (4) Select at most top-k historical test cases with similarity score greater than t_{Test}, and map them to their associated correct patches, (5) Compute an average similarity among selected historical known correct patches, (6) Compute pairwise similarity between every plausible patch for the bug under resolution and the average similarity of known correct patches from the previous step, and (7) Predict a plausible patch as correct if its similarity to the average similarity of correct patches (from step 5) is greater than some threshold t_{Patch}, otherwise, BATS predicts the patch to be incorrect.

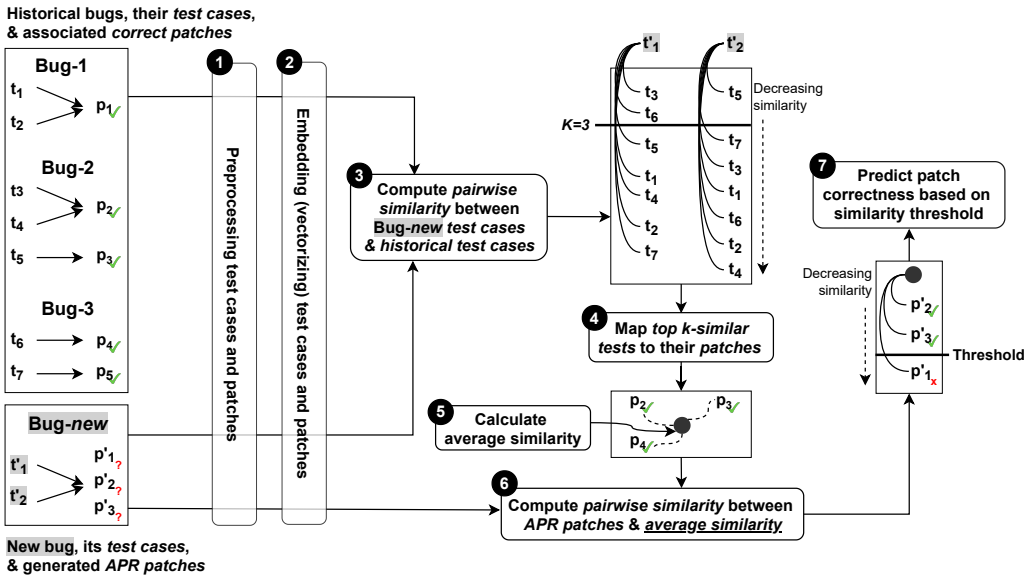

Fig. 1. Overview of BATS.

3.1 Pre-processing Test Cases and Patches

BATS leverages the similarity of test cases and their corresponding patches to predict patch correctness. Therefore, the first two steps of BATS aim at preparing the test cases and patches into a form suitable for computing similarity. One way to achieve this is by embedding patches and test cases into higher dimensional space to obtain numerical vectors that are suitable for vector similarity computations.

Tokenizing test cases. BATS treats the source code of individual test methods as sequences of tokens while also using camelCase tokenization to further breakdown identifiers into their sub tokens.

Tokenizing patches. Patches are tokenized in the same manner as test cases with two differences. First, BATS considers changed lines without their contexts. I.e., it selects added and removed lines only, marked with '+' and '-', respectively. Second, to keep the information about added and removed lines, BATS keeps the '+' and '-' markers as part of each patch line.

3.2 Embedding Test Cases and Patches

BATS relies on similarity calculations between test cases and between patches. Therefore, the second step is to embed test cases and patches into a higher dimensional space to enable similarity computations. An embedding is a function that maps each token into a high dimension real-value vector while maintaining semantic similarities between similar tokens.

In our approach, embedding individual test methods is straightforward but it is not the case for patches. Patches are composed of several individual hunks (contiguous changes) while the order of the hunks is irrelevant to the patch. Each hunk can be embedded as a sequence of tokens into a single vector. But how can we combine the different hunks to obtain a single vector representing the entire patch? Simple concatenation of the vectors of different hunks does not produce a unique vector. Because there is no specific order for the individual hunks, different orderings of the different hunks produces different vectors for the same patch. Therefore, instead of concatenating vectors of

different hunks, we sum the vectors of the different hunks to obtain a unique vector representing the entire patch.

To obtain the embeddings for test cases and patches, we leverage three state-of-the-art pre-trained models:

- **code2vec.** Alon *et al.* [2] leverage the AST representation of a method to produce its embedding. code2vec has been applied to a variety of downstream tasks, including predicting method names, where it revealed its ability to learn the structure and semantics of code fragments. We propose in this work to leverage code2vec for embedding test cases. To that end, we build on a pre-trained model provided by the authors [2] who trained the model on a dataset containing ~14 million Java methods.
- **CC2Vec.** Hoang *et al.* [25] introduced the CC2Vec hierarchical attention neural network model for learning vector representations of patches. In the training phase, the learning is guided by the commit messages that are associated with patches and uses them as semantic descriptions of the patches. We propose in this work to leverage CC2Vec for embedding APR-generated patches. We consider the same architecture where we skip the feature crossing layer and train a new model building on the large dataset of 24,000 patches provided by the authors.
- **BERT.** Another popular embedding method that is applied to natural language is BERT [11], a transformer-based self-supervised language model. Recent work in software engineering fine-tuned BERT on code fragments and applied it to produce embeddings that were shown effective [90, 93]. Tian *et al.* [72] recently leveraged BERT in their experiments on filtering out incorrect patches based on the similarity between buggy code and patched code. For comprehensive experiments, we also propose to investigate using BERT for embedding patches.

3.3 Finding Similar Test Cases

The major hypothesis of BATS is that similar failing test cases have similar associated patches. Therefore, the third step of the approach is to find similar test cases from historical fixed bugs, i.e., test cases that failed before their associated fixes are applied, and are similar to the failing test cases of the current bug under resolution. To this end, BATS computes pairwise similarities between each failing test of the bug under resolution and all tests found in the search space of BATS. To compute the similarity between test cases, we apply the Euclidean distance to their code2vec embeddings. Then using a similarity threshold, t_{Test}, BATS selects at most the top-k historical tests with similarity score > t_{Test}. If the number of top k similar test cases is smaller than the number of tests with similarity score > t_{Test}, k is adjusted accordingly. Note that historical failing test cases do not need to be from the history of the same project of the bug under resolution.

3.4 Mapping Historical Failing Test Cases to their Patches

When BATS finds the most similar test cases that failed in the past, BATS further maps these historical failing tests to their associated correct patches which were applied and accepted as correct fixes for those failing tests. Our hypothesis is that a plausible patch for the current bug under resolution is correct if it is similar to these historical correct patches because their failing test cases are also similar. To facilitate the comparison of the plausible patches to these historical correct patches[1], BATS averages the historical correct patches by computing an *average* of their embedding vectors.

[1]The patches were written by developers to fix the related bugs and were committed in the historical repository of related projects.

3.5 Predicting Patch Correctness

To predict whether a given plausible patch is correct or not, BATS calculates the similarity between this patch and the average of the historical correct patches obtained in the previous step. BATS computes the similarity of patches through Euclidean and Cosine similarity measurements. If this similarity score is higher than a threshold t_{Patch}, BATS predicts that this patch is correct, otherwise, the plausible patch is predicted as incorrect. In our experiments, we set $t_{Patch} = 0.5$.

3.6 An Example

Consider the following example of bug Chart-26 from the Defects4J dataset. Chart-26 triggers 22 test cases to fail. To fix this bug, APR tools SOFix and KaliA generate the two plausible patches presented in Figure 2 and Figure 3, respectively.

```
--- .../source/org/jfree/chart/axis/Axis.java
+++ .../source/org/jfree/chart/axis/Axis.java
@@ -1189,10 +1189,12 @@
         }
         if (plotState != null && hotspot != null) {
             ChartRenderingInfo owner = plotState.getOwner();
+            if (owner != null) {
                 EntityCollection entities = owner.getEntityCollection();
                 if (entities != null) {
                     entities.add(new AxisLabelEntity(this, hotspot,
                             this.labelToolTip, this.labelURL));
                 }
+            }
         }
         return state;
```

Fig. 2. A correct patch generated by APR SOFix for the Defects4J bug Chart-26.

```
--- .../source/org/jfree/chart/plot/CategoryPlot.java
+++ .../source/org/jfree/chart/plot/CategoryPlot.java
@@ -2541,7 +2541,9 @@

         // record the plot area...
         if (state == null) {
         // if the incoming state is null, no information will be passed
+        if (true)
+             return;
         // back to the caller - but we create a temporary state to record
         // the plot area, since that is used later by the axes
         state = new PlotRenderingInfo(null);
```

Fig. 3. An incorrect patch generated by APR KaliA for the Defects4J bug Chart-26.

First, to find out whether any of the two patches is correct, BATS looks for test cases that are similar to the 22 failing test cases in the available search space. The search space includes the history of the Chart project as well as other projects, when available. Second, BATS, thanks to its code2vec-based similarity checker, identifies two test cases as similar to some of the 22 failing tests: one test case is associated with bug Chart-4 and the other is associated with bug Chart-25. A manual investigation of the semantics of these two test cases reveals that they indeed aim at detecting unhandled Null pointer dereferences.

Third, after BATS maps the two identified historical test cases to their corresponding correct patches, it measures the similarity of the APR-generated patches to these relevant correct patches. Finally, based on the similarity score and a similarity threshold, BATS precisely predicts that the generated patches by SOFix and KaliA are correct and incorrect, respectively. When we manually inspect the historical correct patches that were applied to fix Chart-4 (cf. Figure 4) and Chart-25 (cf. Figure 5), we notice that they both implement similar behavior (i.e., adding null check) as the

```
--- .../source/org/jfree/chart/plot/XYPlot.java
+++ .../source/org/jfree/chart/plot/XYPlot.java
@@ -4490,6 +4490,7 @@ public class XYPlot extends Plot implements ValueAxisPlot, Pannable,
                     }
                 }

+                if (r != null) {
                     Collection c = r.getAnnotations();
                     Iterator i = c.iterator();
                     while (i.hasNext()) {
@@ -4498,6 +4499,7 @@ public class XYPlot extends Plot implements ValueAxisPlot, Pannable,
                             includedAnnotations.add(a);
                         }
                     }
+                }
             }
         }
```

Fig. 4. A correct developer-written patch for the Defects4J bug Chart-4.

```
--- .../source/org/jfree/chart/renderer/category/StatisticalBarRenderer.java
+++ .../source/org/jfree/chart/renderer/category/StatisticalBarRenderer.java
@@ -256,6 +256,9 @@ public class StatisticalBarRenderer extends BarRenderer

         // BAR X
         Number meanValue = dataset.getMeanValue(row, column);
+        if (meanValue == null) {
+            return;
+        }

         double value = meanValue.doubleValue();
         double base = 0.0;
@@ -312,7 +315,9 @@ public class StatisticalBarRenderer extends BarRenderer
         }

         // standard deviation lines
-            double valueDelta = dataset.getStdDevValue(row, column).doubleValue();
+        Number n = dataset.getStdDevValue(row, column);
+        if (n != null) {
+            double valueDelta = n.doubleValue();
             double highVal = rangeAxis.valueToJava2D(meanValue.doubleValue()
                     + valueDelta, dataArea, yAxisLocation);
             double lowVal = rangeAxis.valueToJava2D(meanValue.doubleValue()
@@ -341,6 +346,7 @@ public class StatisticalBarRenderer extends BarRenderer
             line = new Line2D.Double(lowVal, rectY + rectHeight * 0.25,
                                      lowVal, rectY + rectHeight * 0.75);
             g2.draw(line);
+        }

         CategoryItemLabelGenerator generator = getItemLabelGenerator(row,
                 column);
```

Fig. 5. A correct developer-written patch for the Defects4J bug Chart-25.

proposed patch by SOFix (cf. Figure2). On the contrary, the KaliA patch (cf. Figure 3) suggests an irrelevant code change.

In our evaluation, there are 17 plausible patches generated by different APR tools for Chart-26[2]. In Figure 6, we see the 17 patches ranked according to their similarity score computed by BATS. On the x-axis, each patch is labeled with a combination of the name of the APR tool that generated it and a numerical id as a tool may generate more than one patch. We note that most of the correct patches (grey bars) are ranked ahead of incorrect patches (white bars) which confirms that our hypothesis is effective in discriminating correct patches from incorrect ones.

[2]These APR-generated patches have been labeled in previous work.

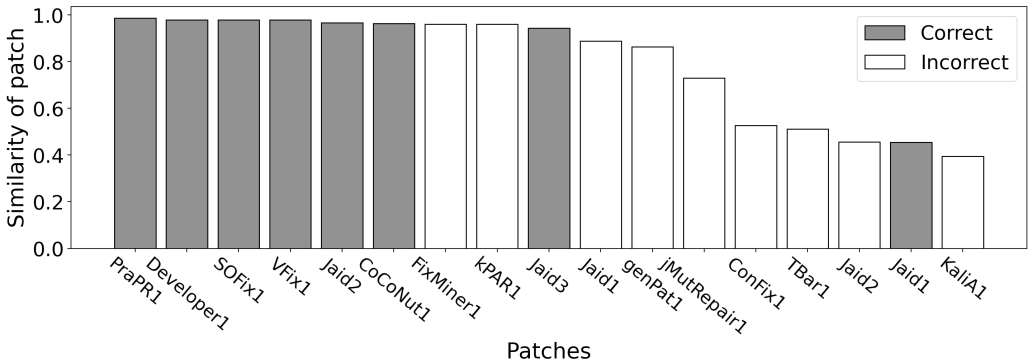

Fig. 6. The ranked patches generated by APR tools for Chart-26. The numerical value next to each tool name indicates patch id since a tool can generate more than one patch.

4 EXPERIMENTAL SETUP

In this section, we introduce the experimental setup to evaluate our hypothesis and our approach, BATS. We present the research questions in Section 4.1, the datasets in Section 4.2, and the evaluation metrics in Sections 4.3 and 4.4.

4.1 Research Questions

Our research questions aim to validate the hypothesis, draw insights for developing a prediction method for patch correctness and finally assess the BATS approach with APR-generated plausible patches while comparing against recent state of the art approaches.

RQ-1. *Does the similarity of bug-triggering test cases correlate with the similarity of the associated bug fixing patches?* This research question aims to validate the feasibility of our proposed hypothesis. To this end, we conduct two experiments: clustering similar test cases and assessing patch similarity with the similar test cases. This offers insights into the design of the patch correctness identification system based on inferred thresholds.

RQ-2. *To what extent can an approach assessing patch behaviour against test specification based on unsupervised learning be effective in identifying correct patches among plausible ones?* With this research question, we first present the implementation of BATS and evaluate its performance in identifying correct patches in 1,278 patches generated by 32 APR tools.

RQ-3. *Can BATS achieve competitive results against the recent state of the art approaches?* In this research question, we compare BATS against static and dynamic approaches of predicting patch correctness for APR tools. Then we explore the possibility of leveraging BATS to complement the state of the art in identifying correct patches.

4.2 Datasets

We focus our experiments on the Defects4J [32] benchmark since it is widely used in the literature, and we can readily collect plausible patches generated by APR tools on the programs included in the benchmark. Table 1 provides the statistics on the collected patches. Besides the 205 developer (correct) patches provided in the benchmark, we also leverage the reproduced dataset from the study of 16 APR systems by Liu et al. [50], which we augment with a dataset provided by Ye et al. [88]. Finally, we also scan the artifacts released in the literature towards identifying plausible patches generated by recent APR tools. Overall, we share with the community the largest dataset[3], to-date, of patches for Defects4J bugs, which includes hundreds of overfitting (i.e., incorrect) patches.

[3]https://github.com/HaoyeTianCoder/BATS

Table 1. Statistics on the dataset of developer-written and APR-generated patches.

Subject	Patches	Incorrect	Correct	Subject	Patches	Incorrect	Correct
Developer [32]	205	0	205	**jGenProg2015** [12]	11	8	3
3sFix [9]	60	57	3	**jKali** [54]	14	12	2
ACS [81]	21	6	15	**jMutRepair** [54]	15	12	3
ARJA [91]	183	168	15	**KaliA** [91]	14	14	0
CapGen [77]	64	39	25	**kPAR** [45]	50	42	8
Cardumen [55]	10	8	2	**LSRepair** [48]	18	15	3
CoCoNut [53]	28	0	28	**Nopol2015** [12]	10	8	2
ConFix [34]	66	52	14	**PraPR** [21]	22	0	22
DeepRepair [78]	13	9	4	**RSRepiarA** [91]	18	18	0
DynaMoth [13]	18	18	0	**SequenceR** [8]	35	24	11
ELIXIR [65]	36	13	23	**SimFix** [30]	35	12	23
FixMiner [36]	27	17	10	**SketchFix** [26]	21	9	12
GenPat [29]	29	18	11	**SOFix** [51]	21	2	19
GenProgA [91]	14	14	0	**ssFix** [79]	19	8	11
HDRepair [40]	6	2	4	**TBar** [47]	57	36	21
Hercules [66]	51	14	37	**VFix** [83]	23	1	22
JAID [7]	64	33	31				
				All	1,278	689	589

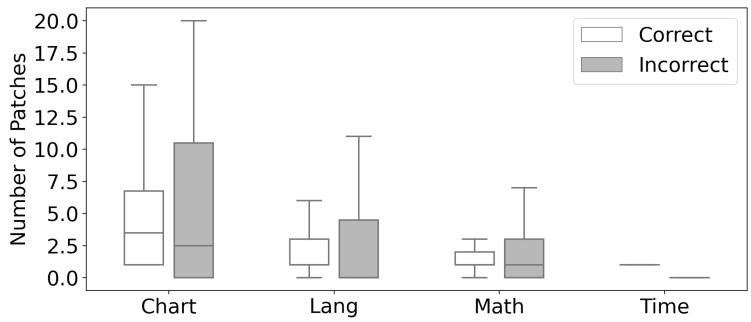

Fig. 7. Distribution of the number of collected patches per project in the Defects4j dataset.

Dataset per experiment: For answering RQ-1, we rely on the 1,120 failing test cases and the associated 205 developer patches in the Defects4J dataset. For answering RQ-2 and RQ-3 (assessment of the BATS approach), we consider the 1,278 plausible patches generated by APR tools or provided by the Defects4j project developers. In these RQs, we also rely on the 1,120 test cases and 205 developer patches as the search space for similar test cases to the failing test case being addressed. We ensure, however, that for every execution of BATS we remove from the search space the failing test cases and the developer patches that are related to the bug under resolution.

Figure 7 presents the distribution of 1,278 generated patches for Chart, Lang, Math, and Time projects that have been widely used in the community of automated program repair [49, 50] and patch correctness identification [72, 80].

4.3 Cluster Analysis Metrics

To group similar test cases and similar patches together respectively, we explore unsupervised learning algorithms to perform clustering. K-means [3] generally provides a good clustering performance and strong interpretability. Its main limitation, however, is that it requires the user to specify the number of clusters k, which is unfortunately specific to the datasets. We therefore

adopt a hierarchical clustering algorithm, e.g., bisecting K-means [33], to empirically determine the appropriate value of k.

Bisecting K-means. This algorithm was initially proposed to overcome the challenges of K-means (local minima, non-spherical clusters, etc.). Bisecting K-means modifies the K-Means algorithm to produce partitional/hierarchical clustering, thus recognizes clusters of any shape and size. The algorithm starts by splitting the dataset into two clusters based on K-means. Then, iteratively, each cluster is split. Each time the two clusters are identified as those presenting the smaller sum of squared errors (SSE). By placing a threshold for the performance in terms of the sum of squared errors, the clustering iterations can be stopped. This algorithm is therefore convenient to infer a reasonable number k of clusters in a dataset.

Cluster Similarity Coefficient (Adjusted Silhouette Coefficient). Once clusters are yielded, we can measure to what extent each element is actually similar to its own cluster (cohesion) compared to other clusters (separation). To that end, we propose a similarity coefficient (SC) and cluster similarity coefficient (CSC) metrics of each cluster element based on its internal similarity and its external similarity towards other elements. We use the following equations:

$$SC(e) = \frac{in(e) - out(e)}{\max\{in(e), out(e)\}} \quad (1) \qquad CSC = \frac{1}{n}\sum_{e=1}^{n} SC(e) \quad (2)$$

where $in(e)$ represents the average Euclidean Similarity from the (test case or patch) embedding e to other embeddings in the same cluster; $out(e)$ represents the average Euclidean Similarity from the embedding e to the embeddings in other clusters. The value range of SC is [-1,1]: the larger the value of SC, the better the clustering effect. When SC value is greater than 0, the clustering is consistent. And in order to measure the overall performance, CSC (averaging SC) is used to calculate the coefficient taking into account all clusters.

Sum of Squared Error (SSE). Finally, we rely on the commonly-used sum of squared errors to measure the variance within clusters. SSE in our study is computed as the sum of the squared differences between each embedding and its cluster's mean. If within each and every cluster, all cases are identical, the SSE would then be equal to 0 as per equation 3.

$$SSE = \sum_{i=1}^{k}\sum_{j=1}^{n_i}\left(x_{ij} - x_i\right)^2 \quad (3)$$

where k represents the number of clusters, n_i represents the number of elements of the i-th cluster, x_{ij} represents the j-th element of the i-th cluster, and x_i represents the center element of the i-th cluster. Therefore, the smaller the SSE, the better the clustering effect.

4.4 Performance Metrics

We consider the **Recall** of BATS in two dimensions:

- **+Recall** measures to what extent correct patches are identified, i.e., the percentage of correct patches that are indeed predicted as correct [72].
- **-Recall** measures to what extent incorrect patches are filtered out, i.e., the percentage of incorrect patches that are indeed predicted as incorrect [72].

$$+Recall = \frac{TP}{TP + FN} \quad (4) \qquad -Recall = \frac{TN}{TN + FP} \quad (5)$$

where TP represents true positive, FN represents false negative, FP represents false positive, TN represents true negative.

Area Under Curve (AUC) and F1. By considering the similarity score as a prediction probability, BATS can be evaluated like any machine learning model with the common metrics such as AUC and F1 score (harmonic mean between precision and recall for identifying correct patches).

MAP and MRR. Since BATS ranks all generated patches based on similarity scores, instead of simply considering the top-1 as the correct, we can consider a recommendation and leverage common metrics used in assessing the ranked list. The mean average precision (MAP) and mean reciprocal rank (MRR) are such metrics that help assess whether BATS can place correct patches ahead of incorrect patches in the ranked list presented to the developers.

$$MAP = \frac{1}{n}\sum_{i=1}^{n}\frac{\sum_{j=1}^{m}(P_{ij} \times Rel_{ij})}{\#\text{ correct patches}} \quad (6) \qquad MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i} \quad (7)$$

Where Rel_{ij} is an indicator function that is 1 if the j-th patch is correct in list i, otherwise it is 0. P_{ij} is the precision at the threshold j in the list i, and the $rank$ represents the first correct ranking.

BATS relies on the unsupervised learning technique and considers to set thresholds of similarities among test cases and patches to predict the correctness of patches. Given that in practice we cannot tune the decision threshold based on the specific case of each bug, we fix a single similarity threshold to compute the accuracy, precision, and false positives. We set the model prediction threshold to 0.5. As soon as the normalized similarity score between the generated patch and the identified cluster of historical patches is higher than 0.5, we predict the patch as a correct one. Otherwise, we predict it as incorrect. We note, however, that in the literature, some approaches [80] are assessed by adjusting the threshold in the test data to achieve the best possible +Recall.

5 EXPERIMENTAL RESULTS

In this section, we first present the validation of the hypothesis in our work (Answer to RQ-1 in Section 5.1). Then , we report the experimental results of patch correctness prediction on our collected dataset (Answer to RQ-2 in Section 5.2 and RQ-3 in Section 5.3).

5.1 [RQ-1] Cluster of Similar Test Cases and Patches

[Objective]: We perform experiments to answer RQ-1, whether the proposed hypothesis is valid for patch correctness identification with the following two sub questions to observe whether failing test cases are similar when the associated patches are similar. First, we investigate whether the 1,120 failing test cases in Defects4J can be grouped in clearly separable clusters. Based on each such test cluster, we decide that the associated 205 developer patches constitute a cluster. Then, we seek to validate the feasibility of leveraging similar test cases to predict the patch correctness with Defects4J dataset.

- **RQ-1.1** *Do patches cluster well together when their test cases are similar?* To answer this RQ, we automatically cluster test cases into groups of similar test cases, then we assess whether the associated patches in each group also have a good clustering cohesion.
- **RQ-1.2** *Given two test cases, can their similarity score be used as an indicator of their relatedness in terms of patch similarity?* We investigate test case similarity vs. patch similarity hypothesis in a fine-grained manner beyond the clusters.

[Experimental Design for RQ-1.1]: First, we use the code2vec and CC2Vec pre-trained models to produce embeddings for each test case and for each patch respectively in the Defects4J dataset. To cluster test cases, we rely on the bisecting K-means algorithm to produce hierarchical clusters. At each iteration of bisecting K-means, we compute the sum of squared error of the clusters. We use the evolution of reduction in SSE values to decide on a threshold for the number of clusters into which we can split the test cases in our dataset. Experimentally, we observed that the number of clusters

k for which the SSE saturates, i.e., no longer drops, is 40. Additionally, we empirically validate the consistency of our proposed hypothesis by investigating the Cluster Similarity Coefficient (CSC) of different cluster settings (i.e., $k \in \{30, 40, 50\}$).

In this experiment, we leverage the Similarity Coefficient (SC) and Cluster Similarity Coefficient (CSC) (cf. Section 4.3) to assess the consistency and cohesion of the yielded clusters. Nevertheless, we can observe the clustering effect for test cases by investigating the distance distribution of each test case to the center of its cluster: we first consider the distance[4] with all test cases in a single group. Then, we compute the distance of each test case to the center of its K-means-inferred cluster. We consider that if the distribution of distances shows that, on average, distances in a cluster are lower than in the whole group, then the test cases have been well-separated. We confirm that when considering the whole dataset, the distances are the longest (i.e., the median value of distance distributions for all clusters are lower than the one for the whole dataset). In several inferred clusters, the median distance (of the test case to the cluster center) is halved. These observations suggest indeed that the test cases could be readily clustered.

[Experimental Results for RQ-1.1]: Considering the test cases clusters, we infer associated groupings of patches that address these test cases. We then compute the CSC to evaluate the consistency and cohesion of the patch clusters that we have derived. These metrics evaluate the distances among patches within each cluster and the distances among clusters. We validate the metrics on the overall Defects4J ground-truth dataset.

Table 2 presents the cohesion and consistency metrics for the patches regrouped based on the clusters of test cases when the number of clusters is 30, 40 and 50, respectively. The Cluster Similarity Coefficient(CSC) is the average value of similarity coefficient (SC) values for all clusters. "**Qualified**" represents the ratio of clusters that have $SC > 0$ out of all clusters identified. We compare those metrics (on test case clusters) to the associated patch clusters. The positive values of the CSC indicate that, on average, the elements inside the same group are indeed more similar among themselves than they are similar to the elements in other groups. When test cases are well grouped ($CSC > 0$), the corresponding clustering of the associated patches clusters together consistently ($CSC > 0$). In more details, a large ratio of clusters (33/40) also have high cohesion. When adjusting the number of clusters to 30 or 50, the results show that the major clusters of patches (23/30, 37/50) still keep high cohesion, and the clustering of test cases and patches are consistent to each other ($CSC > 0$).

Table 2. Statistics on the performance of clustering of test cases and patches with 30, 40 and 50 clusters.

Subjects	Cluster Similarity Coefficient	Qualified
Test Cases	0.19	30/30
Patches	0.16	23/30
Test Cases	0.19	40/40
Patches	0.16	33/40
Test Cases	0.21	50/50
Patches	0.14	37/50

Figure 8 further presents the similarity coefficient (SC) of test cases and patches for each cluster when k is set to 40. We observe that, for most pairwise clusters of test cases and patches, when the SC value of test cases (presented with grey bar) in one cluster is high, the associated patches (presented with white bar) also have a high SC score. We further calculate a Pearson correlation between the clusters of test cases and the clusters of associated patches, of which value is 0.883

[4]Distance and similarity are two concepts that are used interchangeably in this section depending on the context to facilitate comprehension.

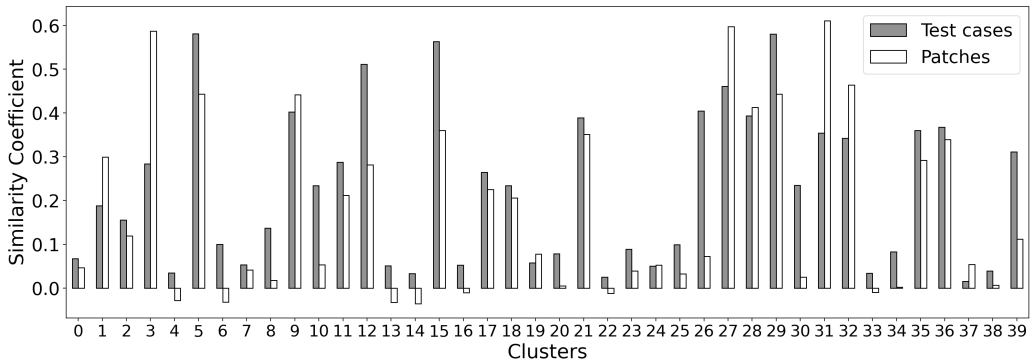

Fig. 8. Similarity coefficient of test cases and patches at each cluster.

> 0. Pearson correlation can be used to measure the linear correlation between two sets of data where a higher positive value (i.e., > 0) indicates a more positive association. Such results indicate that the similar test cases can lead to similar patches.

[RQ-1.1] *Given a cluster of similar test cases, their associated patches cluster consistently. This experiment also hints that the representation models for test cases and clusters yield meaningful embeddings for investigating the relatedness of patches and test cases.*

[Experimental Design for RQ-1.2]: The clustering experiment for RQ-1.1 focuses on average distances within clusters, we further seek to validate the possibility of using test case similarity as a potential heuristic to predict correct patch behavior. The objective is to answer "*whether the similarity of two test cases can be used as an indicator of their relatedness in terms of patch similarity*". To this end, we first consider finding the most similar test case from the search space of historical test cases for the failing-executed test case of a given bug. We then assess to what extent the patch of a given bug is similar to the patch associated to the most similar test case (referred to ① **Scenario H**), of which results are compared against the results in ② **Scenario N** where we compute the average similarity between a given patch and all other patches. The experiments finally investigate scenarios where the closest test case is sought within the all project or only in other projects (excluding the one where the test case is found).

[Experimental Results for RQ-1.2]: Figure 9 presents the overall similarity distribution between each of the 1,120 test cases in the dataset and its closest counterpart: while some test cases indeed have very similar counterparts, many test cases have low similarities with their closest counterparts. These relatively low similarities for many test cases can be explained by the limited number of test cases considered in the study datasets. This results suggest that it may not always make sense, for a given test case, to blindly consider the most similar counterpart since this counterpart can still be highly dissimilar. We thus propose to experimentally determine a threshold to decide when in practice the closest test should not be considered as similar.

In Figure 10, we compare the distributions of the similarity scores for the two scenarios H and N. In all projects, the distributions in the scenario H of our hypothesis present higher similarities. This indicates that the most similar test case is a good proxy to identify a patch[5] that will be more similar than the average patch in a dataset.

In the aforementioned experiment, the test cases in search space are allowed from the same project due to the lack of test cases. To evaluate our hypothesis in the scene of insufficient test cases, we further reproduce the comparison by focusing on test cases and patches that are from

[5]This patch is the one associated with the similar test case that failed in the past.

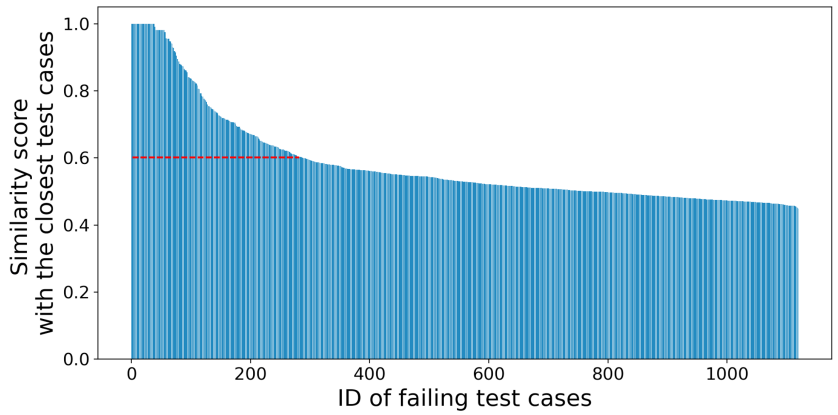

Fig. 9. Distribution on the similarities between each failing test case of each bug and its closest similar test case.

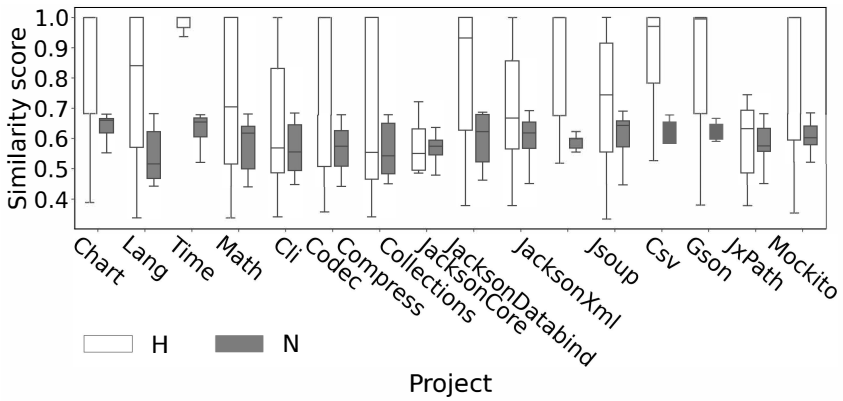

Fig. 10. Distributions on the similarities of pairwise patches (similar patch selected with Scenario H vs. Scenario N from **all projects**, i.e., the search space for searching similar cases is all projects in the dataset).

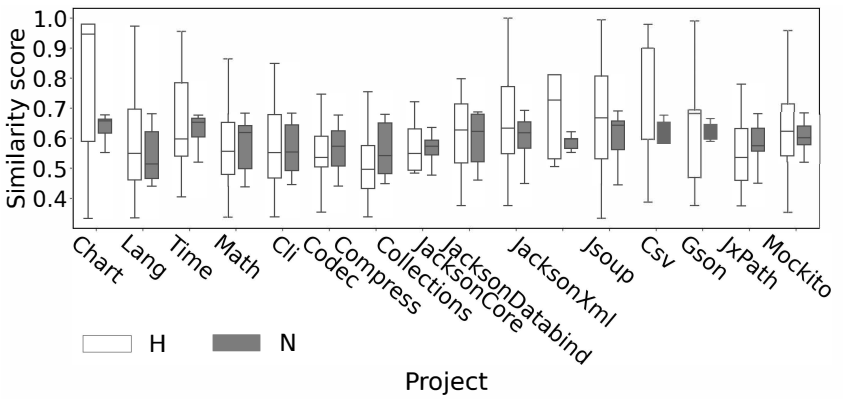

Fig. 11. Distributions on the similarities of pairwise patches (similar patch selected with Scenario H vs. Scenario N from **other projects**, i.e., the search space for searching similar cases does not include the buggy project itself).

other projects (i.e., the buggy project itself is excluded from the search space of test cases). Figure 11 further provides the distributions for the two scenarios H and N. In this case, we note that the

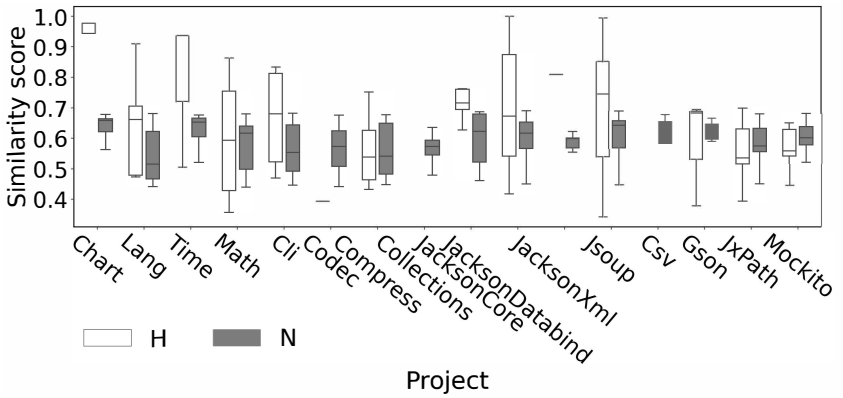

Fig. 12. Distributions on the similarities of pairwise patches (similar patch selected with Scenario H vs. Scenario N from **other projects**, i.e., the search space for searching similar cases does not include the buggy project itself, by setting the threshold at 0.6).

difference is less pronounced for most projects. We postulate that this is due to the fact that similarity scores are low. Thus we propose to set a threshold and consider, for the scenario H, cases where the test cases present a higher similarity than the threshold.

When considering each of the 1,120 test cases, for many of them the closest test case in the search space is actually not a "similar" one. By looking at the pairwise similarity distribution shown in Figure 9, we note that 74% (831/1120) similarity values are lower than 0.6. We therefore arbitrarily decided to use this value as the threshold[6] to explore the hypothesis by isolating the minority of cases where the similarities are significant (more experiments with different thresholds are presented in Section 5.2 for RQ-2). Figure 12 presents the comparison of similarity distribution when the patch pairs are selected with Scenarios H and N from other projects, after setting a threshold of 0.6 for the similarity of test cases in Scenario H to reduce noise. We observe that scenario H now provides the highest similarities for the paired patches (based on the similarities of test cases). Note that some white boxes (scenario H) in the plot are missing is due to lack of high enough similar test cases.

[RQ-1.2] *Given a test case and its most similar test case, their associated patch pair will exhibit a similarity that is statistically higher than the average similarity for all pairs of patches in the same project. This finding is also confirmed across projects.*

5.2 [RQ-2] Identifying Correct Patches with BATS

Objective: Findings in answering the above RQ confirm our hypothesis that test case similarity correlates with patch similarity. BATS is therefore implemented to explore this hypothesis scenario in an APR setting where generated plausible patches (for a failing test case T) are ranked based on their similarity with a set of historical patches that were applied by developers (to address failing test cases similar to T). To answer this RQ, we leverage the 1,278 plausible patches which are composed of 205 developer-written patches and 1,073 APR-generated patches by the tools in Table 1.

By assessing BATS on the collected dataset of plausible patches generated by literature APR tools, our main aim is to demonstrate to the community the feasibility of the proposed research direction for patch assessment.

[6]Note that it aims to validate our hypothesis but not to infer a specific/adaptable threshold.

[Overall Assessment]: We first design a baseline with a simple hypothesis which considers that a patch is more likely to be correct if it is similar to some historically correct patches (i.e., any correct patches). In contrast to BATS, this baseline does not consider failing test cases as the constraint for reducing the search space. We recall that the performance of BATS, which relies on search, is dependent on the availability (in project repositories) of test cases that are actually similar to the failing test case addressed by the APR-generate patches. As introduced earlier, the closest test cases in the search space may actually not be that similar: this is a classical challenge of search engines [43]. Therefore, we propose to filter in only test cases that present a sufficient level of similarity with the targeted test cases. Experimental evaluations will further offer insights on the use of such a threshold.

We chose the cosine similarity for the baseline and BATS implementation. Table 3 reports the classification performance of the baseline (AUC less than 0.6). For reference, the performance of BATS is provided later (cf. Table 4). We note that the baseline performance is similar with BATS when the test similarity threshold is not set. However, when we only consider test cases with higher similarity with the failing test cases, the performance of BATS increases (up to 0.85 for +Recall and 0.71 AUC). CC2Vec, as an embedding model for patches, helps achieve high AUC, F1, +Recall (i.e., the recall in identifying correct patches) and -Recall (i.e., the recall in filtering out incorrect patches).

Table 3. Baseline's performance on identifying (in)correct patches.

Patch embedding†	Correct	Incorrect	AUC	F1	+Recall	-Recall
CC2Vec	589	689	0.586	0.579	0.705 (415)	0.379 (261)
Bert			0.593	0.558	0.647 (381)	0.428 (295)

Table 4. BATS's performance on identifying (in)correct patches.

Patch embedding†	T*	# Correct patches	# Incorrect patches	AUC	F1	+Recall	-Recall
CC2Vec	0.0	589	689	0.557	0.549	0.628 (370)	0.437 (301)
	0.6	144	181	0.559	0.505	0.562 (81)	0.470 (85)
	0.7	94	141	0.678	0.590	0.766 (72)	0.447 (63)
	0.8	57	57	0.718	0.722	0.842 (48)	0.509 (29)
	0.9	41	44	0.709	0.693	0.854 (35)	0.432 (19)
BERT	0.0	589	689	0.561	0.518	0.593 (349)	0.406 (280)
	0.6	144	181	0.611	0.576	0.694 (100)	0.431 (78)
	0.7	94	141	0.639	0.570	0.766 (72)	0.440 (62)
	0.8	57	57	0.676	0.626	0.719 (41)	0.421 (24)
	0.9	41	44	0.647	0.600	0.732 (30)	0.341 (15)

†Embeddings of test cases are always done with code2vec.

*T: Threshold of test case similarity. Given the failing test case of an APR-generated patch, we consider only historical test cases with the similarity which are higher than the threshold. Thus, depending on the threshold, some generated patches cannot be assessed as we are not able to associate them with any past test case.

"(#)" in last two columns represents the number correct/incorrect patches identified by BATS.

Table 5 provides performance results in terms of MAP and MRR. The high metric values further confirm that most correct patches are indeed ranked higher in the recommended patch list that is sorted based on their similarities with historically-relevant patches (given test case similarity). The

Table 5. BATS's performance on ranking correct patches.

Threshold of test case similarity	0.00	0.60	0.70	0.80	0.90
MAP	0.62	0.63	0.71	0.80	0.70
MRR	0.63	0.65	0.74	0.81	0.75

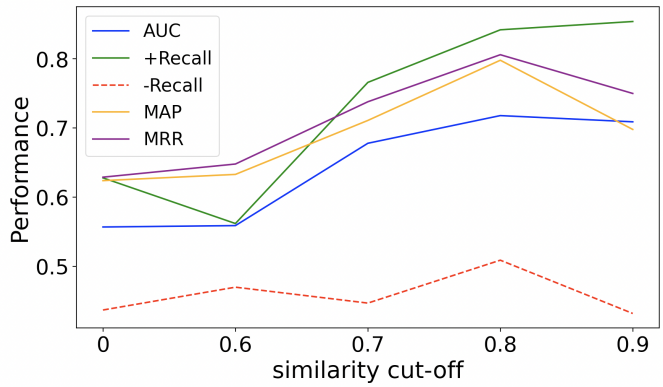

Fig. 13. Performance evolution of BATS with varying threshold of the test-case similarity.

MAP and MRR of the baseline are both 0.63, underperforming against BATS (up to 0.80 and 0.8 for MAP and MRR respectively).

Figure 13 illustrates the overall performance evolution of BATS when the threshold of the test case similarity is varied. We note, while the -Recall does not change drastically, there is a positive effect on +Recall (and other metrics) when the similarity threshold is increased. These results confirm that the underlying hypothesis of BATS is just: when BATS identifies highly similar test cases, its prediction of correctness for APR-generated patches is more accurate.

Representation learning vs. raw strings Beyond the performance metrics on AUC, +/-Recall and F1 of BATS, we propose to investigate the choice of representation learning for embedding patches. To assess the value of leveraging deep learning models for embedding patches, we consider computing similarity of patches as raw strings. To that end, we simply use the Levenshtein distance between the patches (considered as strings, and not after computing embeddings): by setting the threshold of test case similarity as 0.8, we obtain the lowest AUC and F1 values, respectively at 0.46 and 0.53. -Recall even drops at 0.12. These results validate our decision to leverage deep representation learning models for producing patch embeddings.

[RQ-2] *BATS performs well in identifying patch correctness with an AUC ~ 0.7 and an F1~0.7 while the Recall of identifying correct patches reaches ~0.8. Further experimental investigations, with constraints on test case similarities, support our hypothesis: similar test cases are addressed by similar patches.*

5.3 [RQ-3] Competitive/Complementary to the State-of-the-art

Objective: In previous research questions, we validate our hypothesis and develop the pipeline BATS to identify the correctness of patches generated by APR tools. The experimental results show BATS achieves promising performance. To further evaluate BATS in practice, we compare it against the state of the art static and dynamic approaches with the same dataset used for RQ-2. Recall that, in practice, the performance of BATS, is impacted by the availability to find (really) similar test cases. As illustrated in the results of Table 4, when the threshold of test case similarity is set high, the number of patches for which similar test cases are found in our dataset decrease significantly.

While this does not contradict the hypothesis underlying BATS, it may limit its value when larger datasets are not available. Nevertheless, we postulate that BATS can be complementary to the previous two state of the art approaches (i.e., Tian et al. [72] and PATCH-SIM [80]).

5.3.1 **Comparing Against the State-of-the-art.** We conduct the comparison of BATS against the state-of-the-art patch correctness predicting tools, i.e., Tian *et al.*'s deep learning approach [72], PATCH-SIM [80] and ODS [87].

BATS vs. Deep learning approach. Since BATS leverages pre-trained DL-based (deep learning representation) models, we proceed to compare it against the recent related work by Tian *et al.* [72] where DL-based embeddings of patches are leveraged for static checks. To ensure that the results of BATS and of the DL-based approach are compared on the same dataset, we focus on the 114 patches (threshold>0.8) as the testing set of the DL-based approach. Because the approach of Tian *et al.* require a training dataset for supervisely producing a classifier, we use the remaining 1164 (1278-114) patches. As for classification algorithms, we leverage both Logistic Regression (LR) and Random Forrest (RF), following the experiments of the authors. As shown in in Table 6, BATS can recall (+Recall) more correct patches while preserve the same or higher -Recall. It should be noted also that, unlike the DL-approach by Tian *et al.*, BATS doesn't require large dataset for training a classifier.

Table 6. Comparison with a state of the art supervised classifier [72].

Classifier	AUC	F1	+Recall	-Recall
Tian et al. (LR)	0.72	0.68	0.77	0.51
Tian et al. (RF)	0.70	0.49	0.75	0.46
BATS	0.72	0.72	0.84	0.51

BATS vs. PATCH-SIM. The state of the art in dynamic assessment of patch correctness is PATCH-SIM [80]. It targets excluding incorrect patches via comparing execution behaviour of tests for the patched and original programs. We apply PATCH-SIM to the 114 test data to generate prediction. However, PATCH-SIM fails to produce prediction results for some of the bugs/patches[7]. Furthermore, we observed that PATCH-SIM takes several hours to produce a prediction outcome for some patches. In our experiment, we set a one hour timeout for the prediction per patch. Eventually, 48 out of 114 could be evaluated by PATCH-SIM. Our experiment was conducted on Linux server equipped with 8 cores 2.10GHz CPU and 125G memory. Results in Table 7 show that PATCH-SIM achieves a 0.80 of +Recall and a 0.42 of -Recall. Among patches evaluated, the average prediction time for each patch is 1044s (i.e., more than 17 minutes), while BATS only spends ~0.3 second on validating each patch. Overall, the dynamic approach, PATCH-SIM, is constrained in terms of resource requirements, as it needs to generate new test inputs and exploit the behavior similarity of test case executions for validating each patch. The resource cost of BATS is mainly decided by two aspects: ❶ the model training process, although BATS leverages the pre-trained models, and ❷ the search space of historical test cases. The time cost of finding similar test cases will be sharply increased only when the search space is expanded.

Table 7. Comparison with a state of the art dynamic-based patch assessment [80]

Classifier	AUC	F1	+Recall	-Recall
PATCH-SIM	0.61	0.52	0.80	0.42
BATS	0.72	0.72	0.84	0.51

[7]We reported the problem to the PATCH-SIM authors and we are still waiting for their response.

BATS vs. ODS. We also compare our performance against a recent machine learning-based approach leveraging manually-engineered features. We compare BATS against a recent work by Ye *et al.* [87] where the authors propose a supervised learning approach ODS that explores manually engineering patch features for overfitting detection. To predict patch correctness, ODS first constructs the engineering features from the AST representation of patches to express potential behavior information. Such features are used by ODS to train a ML-based model to proceed the classification of patch correctness. While we could not fully reproduce their work on our dataset due to the unavailability of their training dataset, we are able to compare our results with the ones presented in their paper since we have test sets of similar size and from the same sources. Overall, BATS and ODS exhibit similar performance metrics. When they tune their learners to have a high +Recall (e.g., 1.00), their -Recall drops (e.g., 0.46, respectively). Our BATS unsupervised learning approach further aims to cope with two challenges with approaches such as ODS: (1) they require large sets of labelled patches to perform supervised learning; (2) it can be difficult to manually explain a prediction classification (e.g., because features that contribute to the classification decision are difficult to track back to the failing test case specification and may not generalize to new data).

[RQ-3] ❶ *When the availability of similar test cases is satisfied, BATS can achieve competitive performance on predicting the correctness of APR-generated patches against the state-of-the-art dynamic and static approaches.*

5.3.2 **Enhancing the State-of-the-art with BATS.** We present experimental results to demonstrate that we can achieve enhanced performance in correct patch identification by using existing (static or dynamic) state of the art approaches in conjunction with BATS.

I. Supplementing a supervised classifier with BATS.

Objective: Given that BATS is fairly accurate when highly similar test cases (with associated historical patches) are available, we propose to build a pipeline where BATS is applied on the subset of bug cases where such test cases exist. For the rest of bugs, the correctness of generated patches is predicted by using a relevant literature classification-based approach proposed by Tian *et al.* [72]. We then compare the performance of this pipeline against the performance yielded when the classifier is used alone on the whole dataset.

Experiments: We set a threshold of the test case similarity at 0.6 (cf. Table 4) to identify which failing test cases are relevant for assessing the added-value of BATS. The dataset is then split into 325 patches for test and 935 for training a patch classifier described in recent literature: we reproduce the work of Tian *et al.* [72] using their provided artefacts which provide a supervised learning model to classify patches based only on embeddings (computed with Bert). As for learner, we leverage alternatively Logistic Regression (LR) and Random Forrest (RF) following the experiments of the authors.

We first compute the performance of the supervised learning classifier alone on the test dataset. Then, we evaluate the combined pipeline (Tian *et al.* [72] + BATS) with the following procedure: BATS is first applied to predict correctness for all patches that are associated to test cases for which historical test cases with a high similarity (≥ 0.9) can be found. 26.1% (85/325) of patches in the test set are then evaluated by BATS. The rest of patches are passed to the trained supervised classifier which does not require test case information.

Results: The results presented in Table 8 show that the combined pipeline can indeed supplement the baseline classifier. +Recall can be improved by up to 5 percentage points while -Recall can be improved by up to 4 percentage points.

Table 8. Supplementing a supervised classifier with BATS.

Classifier	AUC	F1	+Recall	-Recall
Tian et al. (LR)	0.75	0.60	0.53	0.80
Tian et al. (LR) + BATS	0.75	0.62	0.53	0.85
Tian et al. (RF)	0.75	0.59	0.56	0.82
Tian et al. (RF) + BATS	0.75	0.67	0.61	0.82

[RQ-3] ❷ *BATS can supplement a state of the art patch classification system, which : on bug cases where the failing test case is highly similar to historical test cases, BATS can provide more accurate classification.*

II. Complementing test execution based detection.
Objective: Our objective is to assess whether BATS (which statically reasons about test case similarity) can boost PATCH-SIM [80] (which considers dynamic execution behaviour). We consider that BATS value can be confirmed if it can help exclude incorrect patches that PATCH-SIM could not. Therefore, we build on a similar pipeline than in Section 5.3.2, where BATS is applied instead of PATCH-SIM when enough similar test cases can be found.
Experiments: We apply PATCH-SIM to the above 325 test patches in last Section 5.3.2. As for BATS, we use the Defects4J dataset as search space. Note that, as in all experiments, we ensure that the search space does not include test case or patches linked to the assessed generated patch. To achieve reliable performance with BATS, we must consider only cases where highly similar test cases exist in our dataset. Therefore it is possible that our performance measurement could only be computed on a portion of the test set. Finally, PATCH-SIM can successfully produce results for 153 patches, of which 48 patches are applied by BATS when setting the similarity threshold at 0.8. is sufficient to find similar test cases.
Results: Table 9 presents the performance results of PATCH-SIM (alone) and the combination (PATCH-SIM with BATS). We note that, by applying BATS to the subset of patches where similar test cases are available, we are able to improve the overall performance in patch correctness prediction. Theses results demonstrate that BATS can be complementary to a dynamic approach.

Table 9. Supplementing PATCH-SIM with BATS.

Classifier	AUC	F1	+Recall	-Recall
PATCH-SIM	0.62	0.55	0.82	0.43
PATCH-SIM + BATS	0.65	0.59	0.84	0.51

[RQ-3] ❸ *BATS static approach is complementary to the PATCH-SIM [80] dynamic approach. By being able to identify incorrect and correct patches when test cases are sufficient, BATS implementation confirms our initial hypothesis that statically reasoning about similar test cases offers a novel and promising perspective to the assessment of APR-generate patch correctness.*

6 ABLATION STUDY

6.1 Bug types of failing test cases clusters

In this study, we manually check whether bugs, grouped with respect to the test cases, in each cluster are actually similar or not in terms of bug types. We first note that the failing test cases grouped in the same cluster perform similar checks (e.g., date-related checks). Building up on the dissection study of Defects4J bugs by Sobreira*et al.* [70], we further investigate the categories of bugs in each

cluster. We find that test cases in a given cluster are indeed related to bugs in the same category. For instance, test cases triggering Chart-2 and Chart-4 bugs are in the same cluster and the dissection study data indicates that these two bugs are in the same category of `NullPointerException`. However, it is noteworthy that the categories in the dissection proposed by Sobreira*et al.* are too coarse-grained. Several of our clusters therefore relate to the same category. Finally, note that the dissection study was performed in a previous (smaller) version of Defects4J, which does not allow us to provide comprehensive results for our dataset. Nevertheless, the observations that we have made support the framing of our hypothesis that test cases similarity can reflect very well the category of bugs, and hence of the required fixes. In future work, we will consider exploring other possible representations of bugs beyond test cases, such as bug reports.

6.2 Asymmetry of the hypothesis

The experimental results validate the feasibility of our proposed hypothesis: similar test cases can lead to similar patches for fixing associated bugs. We investigate the symmetry of the hypothesis: *Could similar patches be referred to similar test cases*? To this end, we first independently cluster patches with their similarities, and then assess the similarities of the associated test cases in each patch cluster. With respect to independent clustering of patches, the clustered patches achieve a Cluster Similarity Coefficient (*CSC*) of 0.26 and all the groups are qualified. However, the *CSC* for associated test cases groups is 0.03 that is very close to zero and much lower than the *CSC* value of clustered patches. The results indicate that the symmetry of the hypothesis is invalidated, i.e., similar patches cannot fully be mapped to addressing similar failing test cases. It is reasonable, since different bugs can be fixed with similar code change ways which lead to similar patches [44], but the different bugs are triggered by different test cases.

7 THREATS TO VALIDITY

Threats to External Validity. We relied on Bisecting-K-means for the clustering experiments to validate our hypothesis. Other algorithms may reveal different results. We have mitigated a potential bias by using multiple evaluation metrics to exhaustively assess the clusters. We also relied only on Defects4j to ensure that we can collect enough plausible patches from literature APR experiments.

In future work, the community could further investigate other datasets as well as test cases augmentation (through test generation [61, 67, 71] or code search [35]) to enlarge the datasets of patches and test cases.

Threats to Internal Validity A major threat to internal validity is that we manually process patches to build the dataset. We may have introduced some mismatching errors when associating test cases. To mitigate this threat, we publicly release all artefacts for review by the community.

Towards reasoning about code similarity, we rely on code2vec, which parses test cases to deep learning features. Unfortunately, while most projects format their test case specifications as for typical code (such as in Figure 14), the Closure project presents an ad-hoc format where the essential parts of the specification are formatted as a string (see Figure 15): in such cases, code2vec embeddings abstract away the string as a mere argument to a function, rendering the embeddings semantically irrelevant. Therefore, because we leverage off-the-shelf tools to validate our hypothesis, we simply discard Closure bugs, for which future work can investigate specific learners to parse test specification defined in the form of string. Eventually, our experimental evaluation considers 1,278 plausible patches, 598 of which are correct while 689 are overfitting (i.e., incorrect). Overall, the dataset is, to the best of knowledge, the largest set of plausible patches explored in the literature on patch correctness assessment.

```
public void testDrawWithNullInfo() {
   boolean success = false;
      try {
         BufferedImage image = new BufferedImage(200, 100,
         BufferedImage.TYPE_INT_RGB);
         Graphics2D g2 = image.createGraphics();
         this.chart.draw(g2, new Rectangle2D.Double(0, 0, 200, 100), null, null); g2.dispose()
    ;
      }
   catch (Exception e) {
         success = false;
   }
   assertTrue(success);
}
```

Fig. 14. A typical failing test case specification (Chart-26).

```
public void testComplexInlineNoResultNoParamCall3() {
       test("function f(){a();b();var z=1+1}function _foo(){f()}",
              "function _foo(){{a();b();var z$$inline_0=1+1}}");
```

Fig. 15. String-based format for test specification (Closure-49).

Threats to Construct Validity. The used embedding models are pre-trained and some parameter weights may not be adapted to our work. In future work, we could retrain and fine-tune the parameters after collecting large datasets.

8 RELATED WORK

Automated Program Repair. Automated program repair (APR) aims to liberate program developers from the manual debugging burdens by generating patches automatically, and has attracted significant attention as well as achieved promising performance on fixing bugs [22, 49, 58]. Generate-and-validate program repair is one of the commonly studied APR approaches that generates a set of candidate patches and validates the potential patch by checking the patched programs with static analysis or test suites [52]. Such approaches studied in the literature can be generally summarized into three categories: ① Heuristic based approaches, that construct and iterate over a search space of syntactic program modifications to find the adequate patch for the given in a heuristic way [21, 30, 41, 46, 47, 65, 76, 77, 91]; ② Constrain based approaches, building on constraint solving to synthesize transformations for patch generation [57, 60, 81, 84]; and ③ Learning based approaches which explore machine learning techniques to boosting program repair by learning correct code or natural code transformation [23, 31, 42, 48, 53, 78]. These state-of-the-art APR works are challenged by their generated plausible [63] but incorrect patches that are just overfitting the test suites but not really fix the program bugs. This work aims to boost the APR by proposing a new approach that is dedicated to validating the patch correctness for APR tools.

Patch Correctness. The state-of-the-art APR techniques mainly rely on static analysis techniques or test suite validate to the assess the feasibility of patches generated by themselves, which however could lead to overfitting plausible patches that actually do not fix the bugs or even make the patched programs worse [63, 69]. To address this challenge, practitioners have been studying efficient approaches to automate the identification of correct patches. Xiong *et al.* [80] explored similarity of test case executions to heuristically assess the patch correctness, and implemented a patch validation technique, PATCH-SIM, based on a reasonable observation that a correct patch does not tend to change the behaviour of originally passing test cases but will revise the behaviour of originally failing test cases. Gao *et al.* [18] propose to optimize the search space of patch candidates for APR tools by using the crash-freedom as the oracle to discard patch candidates which crash on the new tests. Their results show that the overfiting problem of patch validation in APR could be alleviated. Yang *et al.* [85] also explored the difference between the runtime behaviour of developer's

and APR-generated patches. While the majority of automatic program repair approaches focus on dynamic execution to decide on the correctness of the patches, Csuvik *et al.* [10] have shown the potential for machine learning techniques to be used in the identification of correct patches: by exploiting embedding models such as Doc2vec and Bert on code, they have found that patches that introduce new variables or make many changes are more likely to be incorrect. Ye *et al.* [87] proposed to leverage manually engineered features to predict overfitting via supervised learning. They constructed an overfitting detection system ODS that trains classifiers for patch prediction by extracting code features from AST representation of buggy code and generated patch code. On the other hand, to assess the reliability of patch correctness prediction techniques, Le *et al.* [38] constructed a gold set of correctness labels of APR-generated patches where several different author and automated patch assessment approaches are evaluated.

Representation Learning. Embedding is a key challenge for reasoning about similarity. Initial works on software engineering artefacts have explored models [11, 37] trained on natural language text (such as Wikipedia). BERT is a widely used bidirectional encoder representations model in textual tasks where it outperforms the state of the art by learning to obtain the conditional parameters. Recent works have however proposed specialized architectures to be trained on code. For example, Alon *et al.* [2] proposed code2vec to capture the semantic properties of code snippets by learning distributed representation. Zhangyin *et al.* presented a pre-trained CodeBert [16] that leveraged programming language and natural language data on code tasks. To learn the representation of code changes, Hoang *et al.* [25] proposed an attention-based neural network model CC2Vec with the hierarchical structure to predict the difference between the removed and added code of the patch.

Code Similarity and Code Clone Detection. Finding code that implements similar functionality is what BATS does to find similar test cases and similar patches. DeepSim [92] leverages deep learning on semantic features matrix representing control- and data-flow information to predict whether a given pair of functions implements similar functionality. Fang et al. [14] introduced a new granularity level of the call-callee relationship to capture similar functionality while leveraging word embeddings and graph embeddings to train a deep neural network for the same task.

To detect code clones across different programming languages, CLCDSA [59] extracts 9 syntactic source code features from the ASTs of different pieces of code and uses either Cosine similarity or trained neural network to detect clones. BATS also uses Euclidean distance and Cosine similarity to find similar test cases and similar patches. Alternatively, SLACC [56] uses dynamic analysis and input-output pairs to detect clones across different programming languages. Finally, binary code similarity has been studied extensively [24] with recent approaches also leveraging graph and instruction embeddings [15, 17, 82] to find similar binaries. These approaches for finding similar code across different programming languages and across binaries could benefit BATS in the future by enabling cross-project and cross-language search for similar test cases and similar patches.

9 CONCLUSION

In this work, we propose and investigate a simple yet effective hypothesis for static patch correctness assessment: given a failing test case, any associated generated patch is likely correct if it is similar to patches that were used to address similar failing test cases. We have validated our hypothesis using the developer correct patches and the associated failing test cases in the Defects4J benchmark. To evaluate the potential of this hypothesis in predicting patch correctness, we propose a patch identification system, BATS, to check patch behaviour against test specification based on unsupervised learning. BATS achieves its highest performance when the similarity of test cases is high, which further validates our hypothesis. Compared against state of the art, BATS outperforms

them in identifying correct patches and filtering out incorrect patches. Despite potential issues in the availability of large datasets of projects to search for historical examples (test cases and their associated patches), we demonstrate that BATS can be complementary to both state of the art static and dynamic approaches to predict patch correctness. Overall, BATS offers insights on a promising research direction for patch assessment.

ACKNOWLEDGEMENTS

This work was supported by the funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 949014), the National Natural Science Foundation of China (Grant No. 62172214), the Natural Science Foundation of Jiangsu Province (Grant No. BK20210279), and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06).

REFERENCES

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[3] David Arthur and Sergei Vassilvitskii. 2006. *k-means++: The advantages of careful seeding.* Technical Report. Stanford.

[4] Marcel Boehme. 2014. *Automated regression testing and verification of complex code changes.* Ph.D. Dissertation. National University of Singapore.

[5] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. 2020. Human-in-the-loop automatic program repair. In *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification*. IEEE, 274–285.

[6] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and Precise On-the-fly Patch Validation for All. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 1123–1134.

[7] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 637–647. https://doi.org/10.1109/ASE.2017.8115674

[8] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).

[9] Zimin Chen and Martin Monperrus. 2018. The remarkable role of similarity in redundancy-based program repair. *arXiv preprint arXiv:1811.05703* (2018).

[10] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *Proceedings of the IEEE 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 18–25.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[12] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. 2015. Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset. *CoRR* abs/1505.07002 (2015). arXiv:1505.07002 http://arxiv.org/abs/1505.07002

[13] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop in Automation of Software Test*. ACM, 85–91. https://doi.org/10.1145/2896921.2896931

[14] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA). ACM, 516–527. https://doi.org/10.1145/3395363.3397362

[15] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 480–491.

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[17] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. 2018. VulSeeker-pro: enhanced semantic learning based binary vulnerability seeker with emulation. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 803–808. https://doi.org/10.1145/3236024.3275524

[18] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 8–18.

[19] Xiang Gao and Abhik Roychoudhury. 2020. Interactive Patch Generation and Suggestion. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ACM, 17–18.

[20] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[21] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 19–30. https://doi.org/10.1145/3293882.3330559

[22] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[23] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[24] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Computing Surveys (CSUR)* 54, 3, Article 51 (2021), 38 pages. https://doi.org/10.1145/3446371

[25] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, 518–529.

[26] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 12–23. https://doi.org/10.1145/3180155.3180245

[27] Qing Huang, An Qiu, Maosheng Zhong, and Yuan Wang. 2020. A Code-Description Representation Learning Model Based on Attention. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 447–455.

[28] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 247–258.

[29] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations From Singular Examples via Big Code. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 255–266. https://doi.org/10.1109/ASE.2019.00033

[30] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 298–309. https://doi.org/10.1145/3213846.3213871

[31] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 1161–1173.

[32] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[33] Michael Steinbach George Karypis, Vipin Kumar, and Michael Steinbach. 2000. A comparison of document clustering techniques. In *TextMining Workshop at KDD2000 (May 2000)*.

[34] Jindae Kim and Sunghun Kim. 2019. Automatic patch generation with context-based change application. *Empirical Software Engineering* 24, 6 (2019), 4071–4106. https://doi.org/10.1007/s10664-019-09742-5

[35] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 946–957.

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

[37] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning (JMLR Workshop and Conference Proceedings, Vol. 32)*. JMLR.org, 1188–1196.

[38] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 524–535.

[39] Xuan-Bach D Le, Quang Loc Le, David Lo, and Claire Le Goues. 2016. Enhancing automated program repair with deductive verification. In *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE,

428–432.

[40] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 213–224. https://doi.org/10.1109/SANER.2016.76

[41] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[42] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. DLfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[43] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. 2021. Opportunities and Challenges in Code Search Tools. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–40.

[44] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. 2021. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* 47, 1 (2021), 165–188. https://doi.org/10.1109/TSE.2018.2884955

[45] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 102–113. https://doi.org/10.1109/ICST.2019.00020

[46] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 456–467. https://doi.org/10.1109/SANER.2019.8667970

[47] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42. https://doi.org/10.1145/3293882.3330577

[48] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference ERA Track*. IEEE, 658–662. https://doi.org/10.1109/APSEC.2018.00085

[49] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[50] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 615–627.

[51] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 118–129. https://doi.org/10.1109/SANER.2018.8330202

[52] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering*. IEEE, 702–713.

[53] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 101–114.

[54] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 441–444. https://doi.org/10.1145/2931037.2948705

[55] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: the Cardumen Mode of Astor. In *Proceedings of the 10th International Symposium on Search Based Software Engineering*. Springer, 65–86. https://doi.org/10.1007/978-3-319-99241-9_3

[56] George Mathew, Chris Parnin, and Kathryn T Stolee. 2020. SLACC: Simion-Based Language Agnostic Code Clones. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 210–221.

[57] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. 691–701.

[58] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[59] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 1026–1037. https://doi.org/10.1109/ASE.2019.00099

[60] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 772–781.

[61] Chao Peng and Ajitha Rajan. 2020. Automated test generation for OpenCL kernels using fuzzing and constraint solving. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 61–70.
[62] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient automated program repair through fault-recorded testing prioritization. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. IEEE, 180–189.
[63] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. 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*. 24–36.
[64] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, 10 (2001), 929–948.
[65] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 648–659. https://doi.org/10.1109/ASE.2017.8115675
[66] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*. IEEE, 13–24. https://doi.org/10.1109/ICSE.2019.00020
[67] Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. 2018. Test generation for higher-order functions in dynamic languages. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.
[68] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 390–405.
[69] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543. https://doi.org/10.1145/2786805.2786825
[70] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 130–140.
[71] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2021. Probabilistic Grammar-based Test Generation. *Software Engineering 2021* P-310 (2021), 97–98.
[72] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 981–992.
[73] Foivos Tsimpourlas, Ajitha Rajan, and Miltiadis Allamanis. 2020. Learning to encode and classify test executions. *arXiv preprint arXiv:2001.02444* (2020).
[74] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163* (2017).
[75] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 968–980.
[76] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 364–374. https://doi.org/10.1109/ICSE.2009.5070536
[77] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1–11. https://doi.org/10.1145/3180155.3180233
[78] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 479–490.
[79] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 660–670. https://doi.org/10.1109/ASE.2017.8115676
[80] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. 789–799.
[81] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 416–426. https://doi.org/10.1109/ICSE.2017.45
[82] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer*

and Communications Security. 363–376.

[83] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 512–523. https://doi.org/10.1109/ICSE.2019.00063

[84] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[85] Bo Yang and Jinqiu Yang. 2020. Exploring the differences between plausible and correct patches at fine-grained level. In *Proceedings of the IEEE 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 1–8.

[86] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 831–841.

[87] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering* (2021).

[88] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26, 2 (2021), 1–38.

[89] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering* 23, 5 (2018), 2948–2979.

[90] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, Vol. 34. 1145–1152.

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

[92] Gang Zhao and Jeff Huang. 2018. DeepSim: deep learning code functional similarity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 141–151.

[93] Shufan Zhou, Beijun Shen, and Hao Zhong. 2019. Lancer: Your code tell me what you need. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1202–1205.