The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches

HAOYE TIAN, University of Luxembourg, Luxembourg
KUI LIU*, Huawei, China
YINGHUA LI, University of Luxembourg, Luxembourg
ABDOUL KADER KABORÉ, University of Luxembourg, Luxembourg
ANIL KOYUNCU, Sabanci University, Turkey
ANDREW HABIB, University of Luxembourg, Luxembourg
LI LI, Monash University, Australia
JUNHAO WEN, Chongqing University, China
JACQUES KLEIN, University of Luxembourg, Luxembourg
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg

A large body of the literature on automated program repair develops approaches where patches are automatically generated to be validated against an oracle (e.g., a test suite). Because such an oracle can be imperfect, the generated patches, although validated by the oracle, may actually be incorrect. While the state of the art explores research directions that require dynamic information or rely on manually-crafted heuristics, we study the benefit of learning code representations in order to learn deep features that may encode the properties of patch correctness. Our empirical work investigates different representation learning approaches for code changes to derive embeddings that are amenable to similarity computations of patch correctness identification, and assess the possibility of accurate classification of correct patch by combining learned embeddings with engineered features. Experimental results demonstrate the potential of learned embeddings to empower Leopard (a patch correctness predicting framework implemented in this work) with learning algorithms in reasoning about patch correctness: a machine learning predictor with BERT transformer-based learned embeddings associated with XGBoost achieves an AUC value of about 0.803 in the prediction of patch correctness on a new dataset of 2,147 labeled patches that we collected for the experiments. Our investigations show that deep learned embeddings can lead to complementary/better performance when comparing against the state-of-the-art, PATCH-SIM, which relies on dynamic information. By combining deep learned embeddings and engineered features, Panther (the upgraded version of Leopard implemented in this work) outperforms Leopard with higher scores in terms of AUC, +Recall and -Recall, and can accurately identify more (in)correct patches that cannot be predicted by the classifiers only with learned embeddings or

*Corresponding author.

Authors' addresses: Haoye Tian, haoye.tian@uni.lu, University of Luxembourg, Luxembourg; Kui Liu, brucekuiliu@gmail.com, Huawei, China; Yinghua Li, yinghua.li@uni.lu, University of Luxembourg, Luxembourg; Abdoul Kader Kaboré, abdoulkader.kabore@uni.lu, University of Luxembourg, Luxembourg; Anil Koyuncu, anil.koyuncu@sabanciuniv.edu, Sabanci University, Istanbul, Turkey; Andrew Habib, andrew.a.habib@gmail.com, University of Luxembourg, Luxembourg; Li Li, li.li@monash.edu, Monash University, Australia; Junhao Wen, jhwen@cqu.edu.cn, Chongqing University, China; Jacques Klein, jacques.klein@uni.lu, University of Luxembourg, Luxembourg; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu, University of Luxembourg, Luxembourg.

0004-5411/2022/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

engineered features. Finally, we use an explainable ML technique, SHAP, to empirically interpret how the learned embeddings and engineered features are contributed to the patch correctness prediction.

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, Distributed Representation Learning, Machine Learning, Embeddings, Features Combination, Explanation

ACM Reference Format:
Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F. Bissyandé. 2022. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. *J. ACM* 1, 1 (December 2022), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automatic program repair (APR) [28, 37, 46], the process of fixing software bugs automatically, has gained a huge momentum with the ever increasing pervasiveness of software. While a few APR techniques try to model program semantics and synthesize execution constraints towards producing correct-by-construction patches, they often fail to scale to large programs. Instead, the large majority of APR research [47] focuses on generate-and-validate approaches where patch candidates are generated and then validated against an oracle.

In the absence of precise program specifications, test suites provide affordable approximations that are widely used as the oracle in APR. In their seminal work on test-based APR, Weimer *et al.* [62] consider that a patch is acceptable as soon as the patched program passes all test cases in the given test suite. Since then, a number of studies [51, 54] have explored the *overfitting problem* in patch validation: an automatically generated patch makes the buggy program pass a given test suite and yet it is incorrect w.r.t. the intended program specification. Since test suites only weakly approximate program specifications, a patched program can indeed satisfy the requirements encoded in the test suite yet present a behavior that deviates from what is expected by the developer but not specified in the existing test suite.

Overfitting patches constitute a key challenge in generate-and-validate APR approaches. Recent studies [18, 21, 22, 33–36, 38, 53, 59, 63] on APR systems highlight the importance of estimating the correct ratio among the valid patches that can be found. To improve this ratio of correct patches, researchers explore several directions which we categorize in three groups depending on when the processing to detect correct patches is applied: before, during, or after patch generation:

(1) *Test-suite augmentation:* Yang *et al.* [70] proposed to generate better test cases to enhance the validation of patches, while Xin and Reiss [65] opted for increasing test inputs.
(2) *Curation of repair operators:* approaches such as CapGen [63] demonstrate that carefully-designed repair operators (e.g., fine-grained fix ingredients) can lead to correct patches.
(3) *Post-processing of generated patches:* Long and Rinard [39] introduced some heuristics to discard patches that are likely overfitting.

So far, the state-of-the-art works targeting the identification of patch correctness are based on computing the similarity of test case execution traces [66], or using machine learning to identify correct patches based on engineered static code features [71], pre-trained natural language-based embeddings [8], and source code trained embeddings [57].

This paper. In this work, we extensively study and evaluate how effective are source code embeddings and engineered features in predicting correct patches. For example, which set of features: engineered or learned embeddings yield better performance in predicting correct patches? Can a combination of both kinds of feature achieve higher performance? Our work fills this gap.

This work builds on and extends our previous work [57] in the following manner:

- We examine and compare the effectiveness of *code embeddings*, *engineered features*, and their combination for predicting patch correctness.
- We present an analysis for detecting which kinds of features contribute to the (in)correct prediction of patch correctness.

We investigate in this paper the feasibility of leveraging advances in deep representation learning to produce embeddings for APR-generated patches and their engineered features, that are amenable to reasoning about correctness.

❶ We investigate different representation learning models adapted to natural language tokens and source code tokens that are more specialized to code changes. Our study considers both pre-trained models and the retraining of models.
❷ We empirically investigate whether, with learned embeddings, the hypothesis of minimal changes incurred by correct patches remains valid: experiments are performed to check the statistical difference between similarity scores yielded by correct patches and those yielded by incorrect patches.
❸ We run exploratory experiments assessing the possibility to select cutoff similarity scores between learned embeddings of buggy code and patched code fragments for heuristically filtering out incorrect patches.
❹ We investigate the discriminative power of learned embeddings in a classification training pipeline (that we named Leopard) aimed at learning to predict patch correctness with learned embeddings. We evaluate our and state of the art approaches by applying a 10-group cross validation in a practical perspective. Comparing against the state of the art, Leopard is complementary to them, even outperforms them on filtering out incorrect patches.
❺ We explore the combination of the learned embeddings and the engineered features to improve the performance on identifying patch correctness with more accurate classification, and implement an upgraded version of Leopard, that we named Panther. The exploring examination is supported by our experimental results.
❻ We empirically interpret the cause of prediction behind features and classifiers to help aware the essence of identifying patch correctness with an explainable ML technique SHAP.

The remainder of this paper is organized as follows. Section 2 provides the background of our work, Section 3 introduces our methodology and study design, Sections 4 and 5 cover the experimental results and a discussion, and Sections 6 and 7 discuss related work and conclude the paper.

2 BACKGROUND

This works leverages learning representation and and machine learning techniques to tackle the problem of identifying correct patches among incorrect and plausible APR-generated patches. Additionally, we examine the explainability of ML models used to predict correct patches. The explainability aspect is of high importance to developers applying APR in their workflow. Therefore, we begin by providing the necessary background of the four pillars of our work: (i) patch correctness, (ii) representation learning for code, (iii) engineered features for predicting patch correctness, and (iv) the explainability of ML models using SHAP.

2.1 Patch Plausibility and Correctness

Defining patch correctness is a non-trivial challenge in the APR community. Until the release of empirical investigations by Smith *et al.* [54], actual correctness (w.r.t. the intended behavior of program) was seldom used as a performance criterion of APR systems. Instead, experimental

results were focused on the number of patches that make the program pass all test cases. Such patches are actually only **plausible**. Qi *et al.* [51] demonstrated that an overwhelming majority of plausible patches generated by GenProg [27], RSRepair [50] and AE [61] are overfitting the test suite while actually being incorrect. To improve the practicability of APR systems to generate **correct** patches, researchers have mainly invested in strengthening the validation oracle (i.e., the test suites). Opad [70], DiffTGen [65], UnsatGuided [75], PATCH-SIM/TEST-SIM [66] generate new test inputs that trigger behavior cases which are not addressed by APR-generated patches.

More recent works [8, 71] are starting to investigate static features and heuristics (or machine learning) to build predictive models of patch correctness. Ye *et al.* [71] presented the ODS approach which relates to our study since it investigated machine learning with static features (i.e., carefully hand-crafted features [71]) extracted from Java program patches. The study of Csuvik *et al.* [8] is also closely related to ours since it explores BERT embeddings to define similarity thresholds between buggy and patched code. Their work however remains preliminary (it does not investigate the discriminative power of features) and has been performed at a very small scale (single pre-trained model on 40 one-line bugs from simple programs).

2.2 Distributed Representation Learning

Learning distributed representations have been widely used to advance several machine learning-based software engineering tasks [9, 12, 13, 49, 76]. In particular, embedding techniques such as **Word2Vec** [23], **Doc2Vec** [23] and **BERT** [9] have been successfully applied to different semantics-related tasks such as code clone detection [60], vulnerability detection [48], code recommendation [77], and commit message generation [15].

By building on the hypothesis of code naturalness [1, 14], a number of software engineering research works have also leveraged the aforementioned approaches for learning distributed representations of code [30, 31]. Alon *et al.* [2] have then proposed **code2vec**, an embedding technique that explores AST paths to take into account structural information in code. More recently, Hoang *et al.* [15] have proposed **CC2Vec**, which further specializes to code changes. Our work explores different techniques across the spectrum of distributed representation learning. We therefore consider four variants from the seemingly-least specialized to code (i.e., Doc2Vec) to the state of the art for code change representation (i.e., CC2Vec).

Doc2Vec [23] is an unsupervised framework mostly used to learn continuous distributed vector representations of sentences, paragraphs and documents, regardless of their lengths. It works on the intuition, inspired by the method of learning word vectors [45], that the document representation should be good enough to predict the words in the document. Doc2Vec has been applied in various software engineering tasks. For example, Wei and Li [60] leveraged Doc2Vec to exploit deep lexical and syntactical features for software functional clone detection. Ndichu *et al.* [48] employed Doc2Vec to learn code structure representation at AST level to predict JavaScript-based attacks.

BERT [9] is a language representation model that has been introduced by an AI language team in Google. BERT is devoted to pre-train deep bidirectional representations from unlabelled texts. Then a pre-trained BERT model can be fine-tuned to accomplish various natural language processing tasks such as question answering or language inference. Zhou *et al.* [77] employed a BERT pre-trained model to extract deep semantic features from code name information of programs in order to perform code recommendation. Yu *et al.* [74] even leveraged BERT on binary code to identify similar binaries.

code2vec [2] is an attention-based neural code embedding model developed to represent code fragments as continuous distributed vectors, by training on AST paths and code tokens. Its embeddings have notably been used to predict the semantic properties of code fragments [2], in order, for

instance, to predict method names. Compton *et al.* [7] recently leveraged code2vec to embed Java classes and learn code structures for the task of variable naming obfuscation.

CC2Vec [15] is a specialized hierarchical attention neural network model which learns vector representations of code changes (i.e., patches) guided by the associated commit messages (which is used as a semantic representation of the patch). As the authors demonstrated in their large empirical evaluation, CC2Vec presents promising performance on commit message generation, bug fixing patch identification, and just-in-time defect prediction.

2.3 Engineered Features

Engineered features are carefully designed and selected features which represent and capture important properties of the underlying data. In APR, one possibility is to statically extract those features from the abstract syntax tree (AST) of the buggy code, the AST of the patched path and the related AST edit scripts as proposed by ODS [71].

ODS extract three kinds of features to detect correct patches: (i) Code description features, e.g., kinds of specific operators in patch code and kinds of statements, (ii) Repair pattern features, whether the repair code has specific patterns according to [41], and (iii) Contextual syntactic features, e.g., the types of faulty statements and the types of their surrounding statements. Using these engineered features, ODS trains a series of machine learning classifiers to predict patch correctness. The experimental evaluation on 713 patches shows that ODS can filter out 57% of overfitting patches and exhibits competitive results when compare with state of the art. We adopt ODS engineered features to conduct our study. Because ODS can not steadily generate all the originally designed engineered features in their research for our patches, we consider to mainly use, in our study, two kinds of engineered features generated by ODS[1]: (1) Prophet features (i.e., the re-implementation of Long *et al.*s' work [39]) and (2) the repair pattern (the related operations of transforming the buggy code to patched code).

2.4 SHAP - SHapley Additive exPlanations

SHAP is a unified framework proposed by Lundberg *et al.* [40] to interpret the output of machine learning models. It connects optimal credit allocation with local explanations using the classic Shapley values from the game theory and their related extensions, thus can provide the importance of each feature for certain particular prediction. Through SHAP, the positive and negative effect of features on prediction can be generated, which allow practitioners to understand which behaviors lead to the (in)correct prediction. Besides, SHAP provides the interaction analysis between features to explore how different features are complementary to each other.

3 METHODOLOGY

In this section, we first present the methodology of our study and then we introduce the research questions that we aim to answer using the proposed methodology.

Overall, our goal is to study the effectiveness of different representations of APR-generated patches and codes for the task of predicting which patches are correct. We first investigate a widespread hypothesis that a patch incurring minimal changes is more likely to be correct. To quantify the patch changes, we exploit different code representation learning methods that leverage deep learning techniques to learn features for code. We adapt them to generate the vectors of buggy code and patched code as well as compute the similarity value of vectors. Based on the similarity distribution, we experimentally filter out incorrect APR-generated patches by relying on naively-defined thresholds.

[1]We have received the confirmation from the authors about this bug and the effectiveness of these two kinds of features.

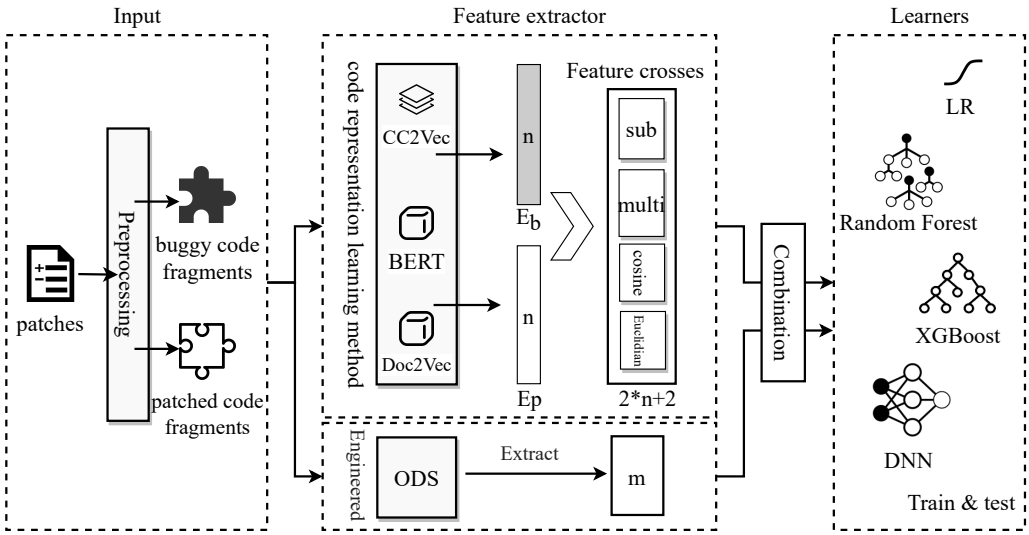


Fig. 1. Overview of PANTHER.

In the view of learning representation reveals the properties of code related to patch correctness, we propose to further identify patch by training classifiers (learners) on the representation vector of a patch. Figure 1 provides an overview of such a pipeline and its variants. To represent patches in a format suitable for learning algorithms, we use the aforementioned representation learning methods to generate vectors for buggy code and patched code. Afterwards, we cross the vectors by applying subtraction, multiplication, cosine similarity and euclidean similarity to obtain the deep learned feature of the patches. The resulting patch embedding has 2*n+2 dimensions where n is the dimension of input code fragment embeddings. The values of the dimension n for BERT, Doc2Vec and CC2Vec are set as 1024, 64 and 64, respectively. On the other hand, we also exploit the manually engineered features that are extracted from the given data, the patch in our case, and aim to capture specific information that is thought to be relevant to the patch correctness. The dimension m for ODS is 195.

Learned and engineered features represent a patch from different perspectives. To improve the identification performance of patch correctness, we further propose three methods (*Ensemble learning*, *Naïve Vector Concatenation*, and *Deep Combination.*) to combine the two features for obtaining the informative representation of a patch. After obtaining a vector that represents a given patch, different machine learning algorithms such as random forest or a deep neural network (DNN) are trained as classifiers that distinguish correct from incorrect APR patches. In the end, we provide the SHAP explanation of the features and interaction of different features that contribute to the patch correctness prediction. Our study follows the four parts below:

(1) The empirical study of a hypothesis in filtering out incorrect patches (RQs 1 to 2),
(2) The effectiveness of machine and deep learning based classifiers with learned representations and engineered features in predicting patch correctness (RQ 3),
(3) The effectiveness of combining learned representations with engineered features in predicting patch correctness (RQ 4), and
(4) The contribution of features in predicting patch correctness (RQ 5).

In the following, we present the details of each research question.

3.1 Research Questions

RQ-1: *Do different representation learning models yield comparable distributions of similarity values between buggy code and patched code?* A widespread hypothesis in program repair is that bug fixing generally induce minimal changes [4, 5, 17, 18, 32, 34, 35, 44, 62, 63, 67]. We propose to investigate whether learned embeddings can be a reliable means for assessing the extent of changes through computation of cosine similarity between vector representations.

RQ-2: *To what extent similarity distributions can be generalized for inferring a cutoff value to filter out incorrect patches?* Following up on RQ1, we propose in this research question to experiment ranking patches based on cosine similarity of their vector representations, and rely on naively-defined similarity thresholds to decide on filtering of incorrect patches.

RQ-3: *Can we learn to identify patch correctness by training predictors with learned embeddings of code input?* We investigate whether deep learned features (i.e., learned embeddings) are indeed relevant for building machine learning predictors for patch correctness. In particular we assess whether such a predictor built with static features can provide comparable performance with dynamic approaches, such as PATCH-SIM, which leverage execution behaviour information. We also compare the performance yielded when using deep learned features against the performance yielded when using the engineered features in the state of the art.

RQ-4: *Can the combination of learned embeddings and engineered features achieve optimum performance for predicting patch correctness?* We investigate the possibility of ensuring high accuracy in patch correctness identification by combining different representations of patches.

RQ-5: *Which features are most useful for predicting patch correctness?* We leverage SHAP explanation models to provide an interpretation of the contribution of different features to the predictions.

3.2 Datasets

We collect patch datasets by building on previous efforts in the community. An initial dataset of correct patches is collected by using five literature benchmarks, namely Bugs.jar [52], Bears [42], Defects4J [19], QuixBugs [29] and ManySStuBs4J [20]. These are human-written patches as committed by developers in open-source project repositories.

We also consider patches generated by APR tools integrated into the `RepairThemAll` framework. We use all patch samples released by Durieux *et al.* [11]. This only includes sample patches that make the programs pass all test cases. They are thus plausible. However, no validation information on correctness was given. In this work, we proceed to manually validate the generated patches, among which we identified 900 correct patches. The correctness validation follows the criteria defined by Liu *et al.* [38]. In a recent study on the efficiency of program repair, Liu *et al.* [38] released a labeled dataset of patches generated by 16 APR systems for the Defects4J bugs. We consider this dataset as well as the labeled dataset that was used to evaluate the PATCH-SIM [66] approach.

Overall, Table 1 summarizes the data sets that we used for our experiments. Each experiment in Section 4 has specific requirements on the data (e.g., large patch sets for training models, labeled datasets for benchmarking classifiers, etc.). For each experiment, we will recall which sub-dataset has been leveraged and why.

3.3 Model Input Pre-processing

Samples in our datasets are patches such as the one presented in Figure 2 extracted from the Defects4J dataset. Our investigations with representation learning however require input data

[2]https://github.com/rjust/defects4j/releases/tag/v2.0.0

Table 1. Datasets of Java patches used in our experiments.

Subjects	contains incorrect patches	contains correct patches	labelled dataset	# Patches
Bears [42]	No	Yes	-	251
Bugs.jar [52]	No	Yes	-	1,158
Defects4J [19]†	No	Yes	-	864
ManySStubBs4J [20]	No	Yes	-	34,051
QuixBugs [29]	No	Yes	-	40
RepairThemAll [11]	Yes	Yes	No‡	64,293
Liu *et al.* [38]	Yes	Yes	Yes	1,245
Xiong *et al.* [66]	Yes	Yes	Yes	139
Total				102,041

†The latest version 2.0.0 of Defects4J[2]is considered in this study.
‡The patches are not labeled in [11]. We support the labeling effort in this study by comparing the generated patches against the developers' patches. The 2,918 patches for IntroClassJava in [11] are also excluded from our study since IntroClassJava is a lab-built Java benchmark transformed from the C program bugs in small student-written programming assignments from IntroClass [26].

```
--- source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java
+++ source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java
@@ -1795,6 +1795,6 @@ public abstract class AbstractCategoryItemRenderer
         int index = this.plot.getIndexOf(this);
         CategoryDataset dataset = this.plot.getDataset(index);
-        if (dataset != null) {
+        if (dataset == null) {
             return result;
         }
```

Fig. 2. Example of a patch for the Defects4J bug Chart-1.

about the buggy and patched code. A straightforward approach to derive those inputs would be to consider the code files before and after the patch. Unfortunately, depending on the size of the code file, the differences could be too minimal to be captured by any similarity measurement. To that end, we propose to focus on the code fragment that appears in the patch. Thus, to represent the buggy code fragment (cf. Figure 3) from the initial patch in Figure 2, we keep all removed lines (i.e., starting with '-') as well as the patch context lines (the code that has not been modified, i.e., those lines not starting with either '-', '+' or '@'). Similarly, the patched code fragment (cf. Figure 4) is represented by added lines (i.e., starting with '+') as well as the same context lines. Since tool support for the representation learning techniques BERT, Doc2Vec, and CC2Vec require each input sample to be on a single line, we flatten multi-line code fragments into a single line.

In contrast to BERT, Doc2Vec, and CC2Vec, which can take as input some syntax-incomplete code fragments, code2vec requires the fragment to be fully parsable in order to extract information on Abstract Syntax Tree paths. Since patch datasets include only text-based diffs, code context is generally truncated and is likely not parsable. However, as just explained, we opt to consider only the removed/added lines to build the buggy and patched code input data. By doing so, we substantially improved the success rate of the JavaExtractor[3] tool used to build the tokens in the code2vec pipeline.

[3]https://github.com/tech-srl/code2vec/tree/master/JavaExtractor

```
1: a/source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java
2:          int index = this.plot.getIndexOf(this);
3:          CategoryDataset dataset = this.plot.getDataset(index);
4:          if (dataset != null) {
5:              return result;
6:          }
```

Fig. 3. Buggy code fragment associated to patch in Fig. 2.

```
1: b/source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java
2:          int index = this.plot.getIndexOf(this);
3:          CategoryDataset dataset = this.plot.getDataset(index);
4:          if (dataset == null) {
5:              return result;
6:          }
```

Fig. 4. Patched code fragment associated to patch in Fig. 2.

3.4 Embedding Models Setup

When representation learning algorithms are applied to some training data, they produce *embedding models* that have learned to map a set of code tokens in the vocabulary of the training data to vectors of numerical values. These vectors are also referred to as *learned embeddings*. Figure 5 illustrates the process of embedding buggy code and patched code for the purpose of our experiments. Considering that the pre-trained embedding models require huge resources (e.g. BERT has 340M parameters to be trained) to fine-tune for our classification task, we resort to directly leverage the pre-trained models to embed the patches, and train the classifiers separately. We propose to use four baseline embedding models from the literature to explore our proposed hypothesis. We consider a variety of models trained on, and targeting, different artifact types (natural language, structured code, code changes).

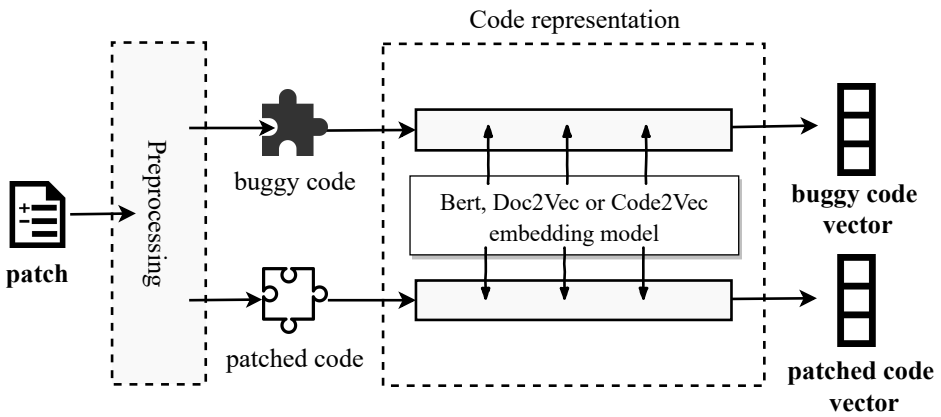


Fig. 5. Producing code fragment learned embeddings with BERT, Doc2Vec and code2vec.

- **BERT.** In the first scenario, we consider an embedding model that initially targets natural language data, both in terms of the learning algorithm and in terms of training data. We thus select BERT, one of the state of the art models, for the evaluation of our hypothesis. The network structure of BERT, however, is deep, meaning that it requires large datasets for training the embedding model. As is now customary in the literature, we instead leverage a pre-trained 24-layer BERT model, which was trained on a Wikipedia corpus.

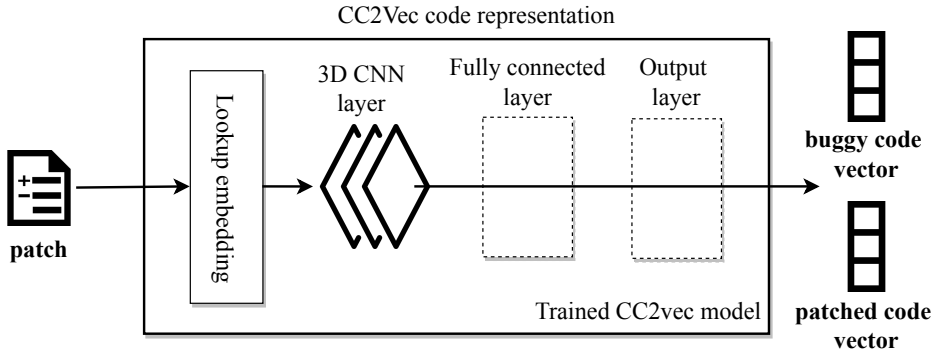


Fig. 6. Extracting code fragment learned embeddings from CC2Vec pre-trained model.

- **Doc2Vec.** In the second scenario, we consider an embedding model that is trained on code data but using a representation learning technique that was developed for text data. Doc2Vec represents documents as a vector by generalizing the basic model word2vec. The code snippets of patches are able to be seen as documents. Therefore, we have trained the Doc2Vec model with code data of 36,364 patches from the 5 repair benchmarks (Bears, Bugs.jar, etc., cf. Table 1).
- **code2vec.** In the third scenario, we consider an embedding model that primarily targets code, both in terms of the learning algorithm and in terms of training data. Code2vec was specifically developed for programming languages and trained on a dataset of 14M methods. On the other hand, CodeBert [13] was trained both on programming and natural language. We thus use in this case a pre-trained model of code2vec, which was trained by the authors using ~14 million code examples from Java projects.
- **CC2Vec.** Finally, in the fourth scenario, we consider an embedding model that was built in representation learning experiments for code changes. CC2Vec [15] models the hierarchical structure of the code change and has been applied to the task of patch identification. However, the pre-trained model that we leveraged from the work of Hoang et al. is embedding each patch into a single vector. We investigate the layers and identify the middle CNN-3D layer as the sweet spot to extract embeddings for buggy code and patched code fragments. Figure 6 illustrates the process.

4 EXPERIMENTS AND RESULTS

We first introduce the metrics used in the experiments. Then, we present the experiments that we designed to answer the research questions of our study. For each experiment, we state the objective, overview the execution details, and present the results.

Our objective is to measure the ability of the approaches in terms of recalling correct patches while filtering out incorrect patches. Thus, we follow the definitions of **Recall** proposed by Tian et al. for the evaluation of their BATS [56] systems:

- **+Recall** measures to what extent correct patches are identified, i.e., the percentage of correct patches that are identified from all correct patches.
- **-Recall** measures to what extent incorrect patches are filtered out, i.e., the percentage of incorrect patches that are filtered out from all incorrect patches.

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

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

Table 2. Datasets used for assessing the similarity between buggy code and correctly-patched code.

	Bears	Bugs.jar	Defects4J	ManySStuBs4J	QuixBugs	Total
# Patches	251	1,158	864	34,051	40	36,364[4]

Accuracy and Precision. The ratio of positive and negative samples of our dataset is balanced (1.3:1). We thus use accuracy and precision to evaluate the performance of the approaches in classifying the patches.

Area Under Curve (AUC) and F1-measure. We train a few machine and deep learning-based classifiers to identify the patch correctness. Therefore, we use two commonly used metrics for evaluating overall performance of the classifiers: AUC and F1 score (harmonic mean between precision and recall for identifying correct patches).

4.1 [RQ-1: Similarity Measurements for Buggy and Patched Code using Embeddings]

Objective: We investigate the capability of different learned embeddings to capture the (dis)similarity between buggy code fragments and the (in)correctly-patched ones. The experiments are performed towards providing answers for two sub-questions:

- RQ-1.1 *Is correctly-patched code actually similar to buggy code based on learned embeddings?*
- RQ-1.2 *To what extent is buggy code more similar to correctly-patched code than to incorrectly-patched code?*

Experimental Design for RQ-1.1: Using the four embedding models considered in our study (cf. Section 3.4), we produce the learned embeddings for buggy and patched code fragments associated to 36k patches from five repair benchmarks shown in Table 2. In this case, the patched code fragment is the correctly-patched code fragment since it comes from labeled benchmark data (generally representing human-written patches). Given those learned embeddings (i.e., deep learned representation vectors of code), we compute the cosine similarity between the vectors representing the buggy and correctly-patched code fragments.

Results for RQ-1.1: Figure 7 presents the boxplots of the similarity distributions with different embedding models and for samples in different datasets. Doc2Vec and code2vec models appear to yield similarity values that are lower than BERT and CC2Vec models.

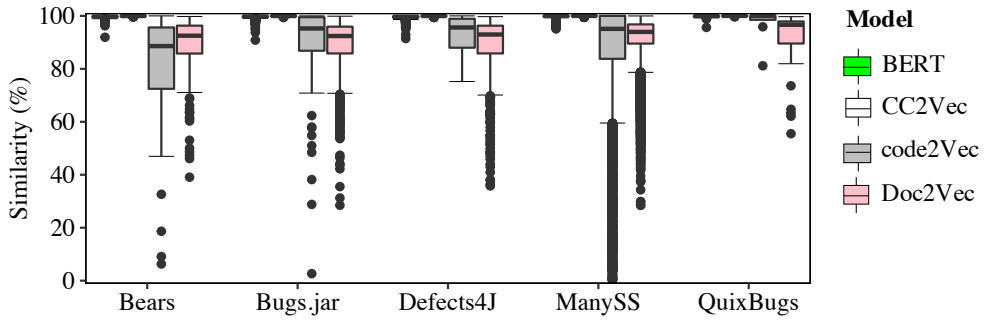
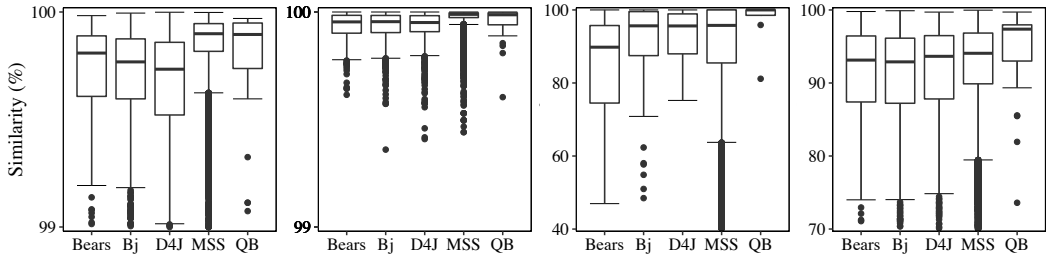


Fig. 7. Distributions of similarity scores between correctly-patched code fragments and buggy ones.

Figure 8 zooms in the boxplot region for each embedding model experiment to overview the differences across different benchmark data. We obverse that, when embedding the patches with BERT, the similarity distribution for the patches in Defects4J dataset is similar to Bugs.jar and Bears dataset, but is different from the dataset ManySStBs4J and QuixBugs. The Mann-Whitney-Wilcoxon (MWW) tests [43, 64] confirm that the similarity of median scores for Defects4J, Bugs.jar and Bears

[4]Due to parsing failures, code2vec learned embeddings are available for 21,135 patches.

is indeed statistically significant. MWW tests further confirms the statistical significance of the difference between Defects4J and ManySStBs4J/QuixBugs scores.



(a) BERT. (b) CC2Vec. (c) code2vec. (d) Doc2Vec.

Fig. 8. Zoomed views of the distributions of similarity scores between correct and buggy code fragments.

Defects4J, Bugs.jar and Bears include diverse human-written patches for a large spectrum of bugs from real-world open-source Java projects. In contrast, ManySStuBs4J only contains patches for single statement bugs. Quixbugs dataset is further limited by its size and the fact that the patches are built by simply mutating the code of small Java implementation of 40 algorithms (e.g., quicksort, levenshtein, etc.).

While CC2Vec and Doc2Vec exhibit roughly similar performance patterns with BERT (although at different scales), the experimental results with code2vec present different patterns across datasets. Note that, due to parsing failures of code2vec, we eventually considered only 118 Bears patches, 123 Bugs.jar patches, 46 Defects4J patches, 20,840 ManySStuBs4J patches and 8 QuixBugs. The change of dataset size could explain the difference with the other embedding models.

✍ **RQ-1.1** ► *learned embeddings of buggy and correctly-patched code fragments exhibit high cosine similarity scores. Median scores are similar for patches that are collected with similar heuristics (e.g., in-the-wild patches vs single-line patches vs debugging example patches). The pre-trained BERT natural language model captures more similarity variations than the CC2Vec model, which is specialized for code changes.*◄

Experimental Design for RQ-1.2: To compare the similarity scores of correctly-patched code fragment vs incorrectly-patched code fragment to the buggy one, we consider combining datasets with correct patches and datasets with incorrect patches. Note that, all patches in our experiments are plausible since we are focused on correctness: plausibility is straightforward to decide based on test suites. Correct patches are provided in benchmarks. However, all the benchmarks in our study do not contain incorrect patches. Therefore, we rely on the dataset released by Liu *et al.* [38]: 674 plausible but incorrect patches generated by 16 repair tools for 184 Defects4J bugs are considered from this dataset. Those 674 incorrect patches are selected within a larger set of incorrect patches by adding the constraint that the incorrect patch should be changed the same code location as the developer-provided patch in the benchmark: such incorrect patch cases may indeed be the most challenging to identify with heuristics.

We consider three scenarios to select correct patches for the comparison of the similarity scores. (1) Imbalanced-all, a quick intuition is that we compare the 674 incorrect patches against all correct patches from 5 benchmarks. (2) Imbalanced-Defects4J, we only use the correct patches from Defects4J. We design the second scenario because the correct patches from other benchmarks may create a sample bias. (3) Balanced-Defects4J, we use the correct patches for the 184 Defects4J bugs that the 674 incorrect patches target. In this scenario, incorrect and correct sets have the same

Table 3. Scenarios for similarity distributions comparison.

Scenario	Incorrect patches	Correct patches
Imbalanced-all[5]	674 incorrect patches	36,364 correct patches from 5 benchmarks in Table 2.
Imbalanced-Defects4J	by 16 APR tools [38]	864 correct patches from Defects4J.
Balanced-Defects4J	for 184 Defects4J bugs.	184 correct patches for the 184 Defects4J bugs.

number of patches. We design this to avoid the underlying bias of imbalanced sets. The comparison is done with different scenarios specified in Table 3.

Results for RQ-1.2: In this experiment, we further assess whether incorrectly-patched code exhibits different similarity score distributions than correctly-patched code. Figure 9 shows the distributions of cosine similarity scores for correct patches (i.e., similarity between buggy code fragments and correctly-patched ones) and incorrect patches (i.e., similarity between buggy code fragments and incorrectly-patched ones). The comparison is done with different scenarios specified in Table 3.

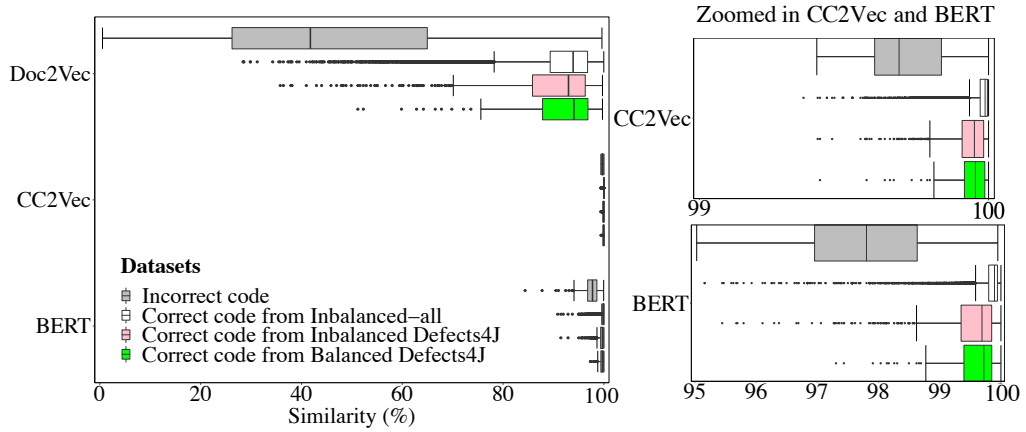


Fig. 9. Comparison of similarity score distributions for code fragments in incorrect and correct patches.

The comparisons do not include the case of learned embeddings for code2vec. Indeed, unlike the previous experiment where code2vec was able to parse enough code fragments, for the considered 184 correct patches of Defects4J, code2vec failed to parse most of the relevant code fragments. Hence, we focus the comparison on the other three embedding models (pre-trained BERT, trained Doc2Vec and pre-trained CC2Vec). Overall, we observe that the distribution of cosine similarity scores is substantially different for correctly-patched and incorrectly-patched code fragments.

We observe that the similarity distributions of buggy code and patched code from incorrect patches are significantly different from the similarities for correct patches. The difference of median values is confirmed to be statistically significant by an MWW test. Note that the difference remains high for BERT, Doc2Vec and CC2Vec whether the correctly-patched code is the counterpart of the incorrectly-patched ones (i.e., the scenario of Balanced-Defects4J) or whether the correctly-patched code is from a larger dataset (i.e., Imbalanced-Defects4J scenarios). As for the comparison with the dataset of Imbalanced-all, the heuristic remains valid but note it may be affected by other benchmarks, i.e., the different bugs caused the results.

[5]Except for Defects4J, there are no publicly-released incorrect patches for APR datasets.

✍ **RQ-1.2** ► *learned embeddings of code fragments with BERT, CC2Vec and Doc2Vec yield similarity scores that, given a buggy code, substantially differ between correctly-patched code and incorrectly-patched one. This result suggests that similarity score can be leveraged to discriminate correct patches from incorrect patches.*◄

4.2 [RQ-2: Filtering of Incorrect Patches based on Similarity Thresholds]

Objective: Following up on the findings related to the first research question, we investigate the selection of cut-off similarity scores to decide on which APR-generated patches are likely incorrect. Results from this investigation will provide insights to guide the exploitation of code learned embeddings in program repair pipelines.

Experimental Design: To select threshold values, we consider the distributions of similarity scores from the above experiments (cf. Section 4.1). Table 4 summarizes relevant statistics on the distributions on the similarity scores distribution for correct patches. Given the differences that were exhibited with incorrect patches in previous experiments, we use, for example, the 1^{st} quartile value as an inferred threshold value.

Table 4. Statistics on the distributions of similarity scores for correct patches of Bears+Bugs.jar+Defects4J.

Subjects	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean
BERT	90.84	99.47	99.73	99.86	100	99.54
CC2Vec	99.36	99.91	99.95	99.98	100	99.93
Doc2Vec	28.49	85.80	92.60	96.10	99.89	89.19
code2vec	2.64	81.19	93.63	98.87	100	87.11

Given our previous findings that different datasets exhibit different similarity score distributions, we also consider inferring a specific threshold for the QuixBugs dataset (cf. statistics in Table 5).

Table 5. Statistics on the distributions of similarity scores for correct patches of QuixBugs.

Subjects	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean
BERT	95.63	99.69	99.89	99.95	99.97	99.66
CC2Vec	99.60	99.94	99.99	100	100	99.95
Doc2Vec	55.51	89.56	96.65	97.90	99.72	91.29
code2vec	81.16	98. 53	100	100	100	97.06

Our test data is constituted of 64,293 patches generated by 11 APR tools in the empirical study of Durieux *et al.* [11]. First, we use the four embedding models to generate learned embeddings of buggy code and patched code fragments and compute cosine similarity scores. Second, for each bug, we rank all generated patches based on the similarity scores between the patched code and the buggy one, where we consider that the higher the score, the more likely the correctness. Finally, to filter incorrect candidates, we consider two experiments:

(1) Patches that lead to similarity scores that are lower to the inferred threshold (i.e., 1^{st} quartile in previous experimental data) will be considered as incorrect. Patches where patched code exhibit higher similarity scores than the threshold are considered correct.
(2) Another approach is to consider only the top-1 patches with the highest similarity scores as correct patches. Other patches are considered incorrect.

In all cases, we systematically validate the correctness of all 64,293 patches to have the correctness labels, for which the dataset authors did not provide (all plausible patches having been considered as valid). First, if the file(s) modified by a patch are not the same buggy files in the benchmark, we systematically consider it as incorrect: with this simple scheme, 33,489 patches are found incorrect.

Table 6. Filtering incorrect patches by generalizing thresholds inferred from Section 4.1.Results.

Dataset		Bears, Bugs.jar and Defects4J		QuixBugs	
# Correct Patches		893		7	
# Incorrect Patches		61,932		1,461	
Model/Metric/Threshold		1st Qu.	Mean	1st Qu.	Mean
BERT	# +CP	57	49	4	4
	# -IP	48,846	51,783	1,387	1,378
	+Recall	6.4%	5.5%	57.1%	57.1%
	-Recall	78.9%	83.6%	94.9%	94.3%
CC2Vec	# +CP	797	789	4	4
	# -IP	19,499	23,738	1,198	1,255
	+Recall	89.2%	88.4%	57.1%	57.1%
	-Recall	31.5%	38.3%	82.0%	85.9%
Doc2Vec	# +CP	794	771	7	7
	# -IP	25,192	33,218	1,226	1,270
	+Recall	88.9%	86.3%	100%	100%
	-Recall	40.7%	53.6%	83.9%	86.9%

"**# +CP**" means the number of correct patches that can be ranked upon the threshold, while "**# -IP**" means the number of incorrect patches that can be filtered out by the threshold. "**+Recall**" and "**-Recall**" represent the recall of identifying correct patches and filtering out incorrect patches, respectively.

Second, with the same file, if the patch is not making changes at the same code locations, we consider it to be incorrect: 26,386 patches are further tagged as incorrect with this decision (cf. Threats to validity in Section 5). Finally, for the remaining 4,418 plausible patches in the dataset, we manually validate correctness by following the strict criteria enumerated by Liu *et al.* [38] to enable reproducibility. Overall, we could label 900 correct patches. The remainders are considered as incorrect.

Results: By considering the patch with the highest (top-1) similarity score between the patched code and buggy code as correct, we were able to identify a correct patch for 10% (with BERT), 9% (with CC2Vec) and 10% (with Doc2Vec) of the bug cases. Overall we also misclassified 96% correct patches as incorrect. However, only 1.5% of incorrect patches were misclassified as correct patches.

Given that a given bug can be fixed with several correct patches, the top-1 criterion may not be adequate. Furthermore, this criterion makes the assumption that a correct patch indeed exists among the patch candidates. By using filtering thresholds inferred from previous experiments (which do not include the test dataset in this experiment), we can attempt to filter all incorrect patches generated by APR tools. Filtering results presented in Table 6 show the recall scores that can be reached. We provide experimental results when we use 1^{st} quartile and Mean values of similarity scores in the "training" set as threshold values. The thresholds are also applied by taking into account the datasets: thresholds learned on QuixBugs benchmark are applied to generated patches for QuixBugs bugs.

✍ **RQ-2** ►*Building on cosine similarity scores, code fragment learned embeddings can help to filter out between 31.5% with CC2Vec and 94.9% with BERT of incorrect patches. While BERT achieves the highest recall of filtering incorrect patches, it produces learned embeddings that lead to a lower recall (at 5.5%) at identifying correct patches.*◄

4.3 [RQ-3: Classification of Correct Patches with Supervised Learning]

Objective: Cosine similarity between learned embeddings (which was used in the previous experiments) considers every deep learned feature as having the same weight as the others in the embedding vector. We investigate the feasibility to infer, using machine learning, the weights that different features may present with respect to patch correctness. To this end, we build a patch correctness prediction framework, LEOPARD (LEarn tO Predict pAtch coRrectness with embeDdings), with the embedding models and machine learning algorithms. We compare the prediction evaluation results of LEOPARD with the achievements of related approaches in the literature. The experiments are performed towards providing insights for the three sub-questions:

- RQ-3.1 *Can LEOPARD learn to predict patch correctness by training classifiers based on the learned embeddings of code ?*
- RQ-3.2 *Can LEOPARD be as reliable as a dynamic state-of-the-art approach such as PATCH-SIM in the patch correctness identification task?*
- RQ-3.3 *To what extent learned embeddings of LEOPARD are providing different prediction results than the engineered features?*

Experimental Design for RQ-3.1: To perform our machine learning experiments, we first require a ground-truth dataset. To that end, we rely on labeled datasets in the literature. Since incorrect patches generated by APR tools are only available for the Defects4J bugs, we focus on labeled patches provided by three independent teams (Liu *et al.* [38], Ye *et al.* [73] and Xiong *et al.* [66]) and other patches generated by APR tools. Very few patches generated by the different tools are actually labeled as correct, which leads to an imbalanced dataset. To reduce the imbalance issue, we supplement the dataset with developer (correct) patches as supplied in the Defects4J benchmark. Note that one developer patch could include multiple fixing hunks for different files, but the extraction of engineered features only work on the patches with respect to changing single file. Thus, we split such patches into sub patches by their changed files to ensure that one sub patch is only involved with one code file. In total, we collect 2,687 patches. After removing duplicates, 2,244 patches are remained. 97 patches are failed to obtain their engineered feature. Eventually, the ground-truth dataset is built with 2,147 patches, shown in Table 7.

Table 7. Dataset for evaluating ML-based predictors of patch correctness.

	Correct patches	Incorrect patches	Total
Liu *et al.* [38]	94	366	460
Ye *et al.* [73]	242	452	694
Xiong *et al.* [66]	30	109	139
Defects4J (developers) [19]	969	0	969
Other APR tools	263	162	425
Dataset	1,598	1,089	2,687
Dataset (deduplicated)	1,288	956	2,244
Dataset (final, with available features)	**1,199**	**948**	**2,147**

Our ground truth dataset patches are then fed to our embedding models in LEOPARD to produce embedding vectors. As for previous experiments, the parsability of Defects4J patch code fragments prevented the application of code2vec: LEOPARD uses pre-trained models of BERT (trained with natural language text) and CC2Vec (trained with code changes) as well as a retrained model of Doc2Vec (trained with patches). Since the representation learning models are applied to code fragments inferred from patches (and not to the patch themselves), LEOPARD collects the embeddings of both buggy code fragment and patched code fragment for each patch. Then LEOPARD must

merge these vectors back into a single input vector for the classification algorithm. We follow an approach that was demonstrated by Hoang et al. [15] in a recent work on bug fix patch prediction: the classification model performs best when features of patched code fragment and buggy code fragment are crossed together.

At first, and following related works in the literature, we used a 10-fold cross validation scheme to evaluate and compare our approach against the state of the art. However, we found that, with this scheme, a patch set generated for the same bug can be split into both the training and testing sets. Such a scenario is actually unrealistic (and biased) since we should not train the model with some labeled patches of a bug that we intend to repair (test set). To address this bias, we propose instead a 10-group cross validation scheme: First, we randomly distribute all bugs into 10 groups. Every group contains unique bugs and their associated patches. Then, we use 9 groups as train data and the remaining group as the test data. Finally, we repeat the selection of train and test groups for ten rounds and obtain the average score of the metrics.

Results for RQ-3.1: We compare the performance of different embedding models using different classification algorithms. Table 8 presents the results with 10-group cross validation setup. All classical metrics used for assessing predictors are reported: Accuracy, Precision, Recall, F1-Measure, Area Under Curve (AUC). XGBoost applied to BERT embeddings yields the best performance on the most of metrics (e.g. AUC with 0.803 and F1-measure with 0.765), while DNN achieves the best performance on precision of 0.744.

Our previous work [57] was conducted through a 5-fold cross validation. To evaluate performance change of the approach on the new augmented dataset, we re-conduct a 5-fold cross validation experiment. The results show that after increasing the number of training examples (1,147 more patches), the performance of the decision tree, logistic regression and naive bayes classifiers are improved. For instance, applying the three classifiers with BERT embeddings, their accuracy, precision, recall and F1-measure are improved with 3 to 23.6 points (except the recall of Naive bayes + BERT embedding is decreased). Their AUC values are increased with 0.067, 0.06, 0.126, respectively. These results provide us the possibility of evolving the patch identification through datasets augmentation. Note that, for the following experiment, we proceed to focus on using 10-group cross validation because of its effectiveness for evaluating the approaches in practice.

✍ **RQ3.1** ► *Tree-based boosting classifiers (Random forest and XGBoost) and Deep learning classifier (DNN) with BERT embeddings yield the promising performance on predicting the patch correctness for APR tools (e.g., F1-measure at 76.5% and AUC at 80.3%).*◄

Experimental Design for RQ-3.2: PATCH-SIM [66] is the state-of-the-art work on predicting the patch correctness for APR tools. It is a dynamic-based approach, which generates execution traces of patched programs with new generated tests, and compares the execution traces across test cases to assess the correctness of APR-generated patches. We propose to apply PATCH-SIM to our collected patches (cf. Table 7). Unfortunately, PATCH-SIM is implemented to run on Defects4J-v1.2.0[6]. Therefore, it failed to process 476 patches generated for some bugs (e.g., JSoup bugs) in the latest version of Defects4J (i.e., Defects4J-v2.0.0). Furthermore, even when PATCH-SIM can run, we observe that it does not yield any prediction output for 1,022 patches[7]. Eventually, we were able to assess the performance of PATCH-SIM on 649 patches. To avoid a potential bias in comparisons, we also conduct the ML-based classification experiments for Leopard on the 649 patches.

[6]https://github.com/rjust/defects4j/releases/tag/v1.2.0

[7]We have reported the issue to the authors but have not yet been made aware of any solution to address it. Note that in their original paper the authors transparently informed readers that the tool indeed is sensitive to the datasets.

Table 8. Evaluation of learned embeddings on six ML classifiers in Leopard.

Learner	Embedding	Accuracy	Precision	Recall	F1-measure	AUC
Decision Trees	BERT	62.1	64.7	70.8	67.6	0.611
	CC2Vec	58.0	61.7	65.5	63.5	0.572
	Doc2Vec	58.7	62.0	67.6	64.6	0.576
Logistic regression	BERT	72.2	73.5	78.7	76.0	0.796
	CC2Vec	61.8	64.8	68.9	66.8	0.679
	Doc2Vec	65.8	66.6	77.7	71.7	0.717
Naive bayes	BERT	66.5	72.5	57.6	65.7	0.726
	CC2Vec	57.6	70.1	31.9	45.7	0.670
	Doc2Vec	55.9	63.0	51.0	56.4	0.610
Random forest	BERT	69.4	68.3	77.9	75.5	0.793
	CC2Vec	62.1	63.9	74.1	68.6	0.705
	Doc2Vec	64.9	63.5	87.6	73.6	0.705
XGBoost	BERT	71.8	71.6	82.1	76.5	0.803
	CC2Vec	65.3	66.4	76.6	71.1	0.729
	Doc2Vec	63.2	63.5	80.2	70.8	0.693
DNN	BERT	70.3	74.4	71.3	72.8	0.767
	CC2Vec	51.8	55.5	69.0	61.6	0.503
	Doc2Vec	63.2	64.7	75.1	69.5	0.679

Table 9. Comparing evaluation of Leopard (BERT embedding + ML classifiers) against PATCH-SIM.

Approach		Accuracy	Precision	Recall	F1-measure	AUC
PATCH-SIM		38.8	24.7	78.9	37.7	0.528
Leopard	BERT + Random forest	41.3	25.5	78.3	38.4	0.594
	BERT + XGBoost	**42.7**	**26.2**	79.6	39.4	**0.614**
	BERT + DNN	40.0	26.1	**85.5**	**40.0**	0.546

Results for RQ-3.2: Table 9 provides the comparing results on predicting patch correctness. In terms of Recall, PATCH-SIM achieved 78.9% that is a bit higher than the BERT embedding + Random forest of Leopard, which demonstrates its ability of recalling correct patch from plausible patches as reported in [66] by its authors. However, the accuracy, precision and AUC measurements are just 38.8%, 24.7% and 52.8%, respectively. These results underperform the three ML classifiers of Leopard. It indicates the many incorrect patches are wrongly identified as correct by PATCH-SIM. Figure 10 further gives an example on comparing the BERT embedding + the XGBoost classifier of Leopard and PATCH-SIM in terms of the number of (in) patches correctly identified by them. XGBoost classifier of Leopard can recall more correct and incorrect patches than the PATCH-SIM, and the 24 correct patches and 124 incorrect patches are exclusively correctly predicted by it.

Time cost. Note that we have recorded that, on average, PATCH-SIM takes ~17.5 minutes to predict the correctness of each patch. In contrast, each of the ML classifiers of Leopard takes less than 1 minute for prediction. However, note that the training of Leopard requires the input of the learned embeddings of patches generated by pre-trained models (e.g. BERT). Such models, which are available on-the-shelf, have been trained using hundreds of TPUs that were run for several hours on a large corpus.

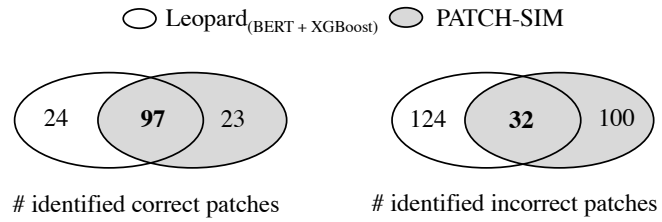


Fig. 10. Comparison on the number of (in)patches correctly identified by Leopard (with the BERT embeddings + the XGBoost learner) against PATCH-SIM.

✍ **RQ-3.2** ► *ML predictors of Leopard trained on learned embeddings can be complementary to the state-of-the-art PATCH-SIM. They can also outperform PATCH-SIM in filtering out more patches generated by APR tools.*

Experimental Design for RQ-3.3: As reported by Ye *et al.* [71] in a recent study, post-processing APR-generated patches through engineered features achieves promising results. Therefore, in this study, we also use some of the engineered features (Prophet features and repair pattern) in[71] to predict correct patches on a larger dataset: overall, our study is based on 2,147 patches while Ye *et al.* applied only 713 patches. Results in this study are given based on 10-group cross validation.

Results for RQ-3.3: Table 10 presents the results of predicting patch correctness with the engineered features. The naive bayes learning algorithm achieves a unusual performance compared to the other five learners. It yields the highest precision, but leads to a much lower recall than others. This suggests that a very small number of correct patches can be recalled via using this learner. The Random Forest and XGBoost learners achieve similarly high performance (e.g., F1-measure at 74.7%/74.1% and AUC at 76.9%/77.6%), and are followed by the DNN learner. Overall, the performance reached with engineered features is generally comparable (in terms of global metrics) to that yielded by Leopard using learned embeddings, except when using the Naive Bayes and Decision Trees learning algorithm.

Table 10. Evaluation of engineered feature on six ML classifiers.

Learner	Accuracy	Precision	Recall	F1-measure	AUC
DecisionTree	66.6	68.6	73.9	71.1	0.666
Logistic regression	70.0	72.7	74.1	73.4	0.773
Naive bayes	49.6	74.6	14.7	24.5	0.689
Random forest	70.7	72.1	77.5	74.7	0.769
XGBoost	70.5	72.6	79.9	74.1	0.776
DNN	69.8	72.1	74.8	73.4	0.777

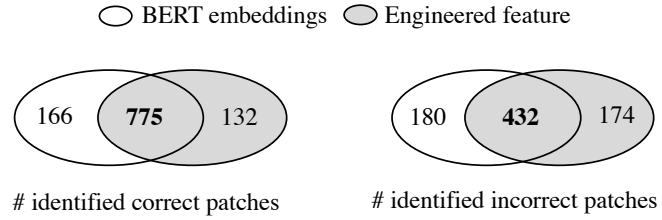


Fig. 11. Comparison on the number of (in)patches correctly identified by the XGBoost classifier with the BERT embeddings and the engineered features.

Figure 11 further illustrates the differences between the XGBoost classifier with the BERT embeddings and the engineered features in terms of the number of identified (in)correct patches.

More (in)correct patches can be correctly identified by the XGBoost classifier with both two scenarios. Nevertheless, there still is a big complementary space of identifying the patch correctness for the two scenarios.

✍ **RQ-3.3** ► *The ML classifiers fed with the engineered features (from static code) can achieve comparable performance to learned embeddings based classifiers in identifying patch correctness. There is nevertheless the possibility to improve the prediction performance in both cases since their correct predictions are not perfectly overlapping: learned embeddings lead to the identification of correct/incorrect patches that are not recalled with engineered features and vice versa.* ◄

4.4 [RQ-4: Combining Learned Embeddings and Engineered Features for more Accurate Classification of Correct Patches]

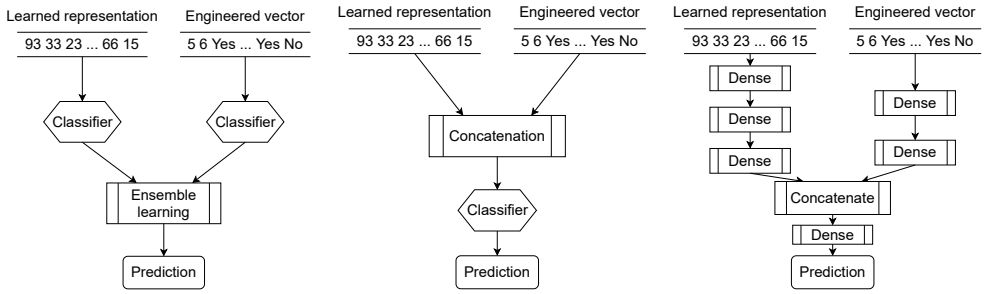
Objective: Following up on the insights from the previous research question, which compared engineered features against learned embeddings, we investigate the potential of leveraging both feature sets to improve the classification of correct patches.

Experimental Design: Leveraging different feature sets can be achieved in several ways, e.g., by concatenating feature vectors or by performing ensemble learning. In this study, we investigate three different methods which are implemented in the upgraded version of Leopard, Panther (Predict pAtch correctNess wiTH the learned Embbeddings and engineeRed features), as illustrated in Figure 12:

(1) *Ensemble learning.* We rely on the six learning algorithms (cf. Tables 8 and 10) to predict the correctness of patches based either on the learned embeddings or on the engineered features. Eventually, to combine both, we simply compute the average prediction probability provided by a pair of classifiers (one trained with learned embeddings and the other with engineered features), and use this probability to decide on patch correctness.
(2) *Naïve Vector Concatenation.* In the second method, we ignore the fact that learned embeddings vectors and engineered feature vectors are not from the same space and propose to Naïvely concatenate them into a single representation. Our intuition, indeed, is that both representations capture different features of patches and can therefore offer, together, a better representation. The yielded concatenated vectors are then used to train the classifiers (with the usual learning algorithms).
(3) *Deep Combination.* In the last method, we consider that learned embeddings and engineered features are from different spaces. Therefore, we must learn their different weights as well as the common representations for them before concatenation. We resort thus to deep neural networks to attempt a deep combination of feature sets before classification.

In this RQ, given the performance of BERT in previous experiments (cf. Table 8), we focus on the BERT embedding model to learn the learned embeddings of patches. Similarly, we only consider Random forest and XGBoost as the best learners to be applied (cf. Table 8 and Table 10). The *Deep Combination* method is based on the work of Cheng *et al.* [6] who proposed a deep learning fusion structure which combined layers that were specialized to explore memorization and generalization of features. Following up this idea of fusion, we design a Double-DNN-fusion structure where learned embeddings are considered useful for generalization and engineered features are considered for memorization. Eventually, we conduct 10-group cross validation for the experimental assessment.

Results: Table 11 presents the performance comparison for correctness identification when using combined features vs using single feature sets. The comparison is done in terms of three main metrics: +Recall (to what extent correct patches can be identified), -Recall (to what extent



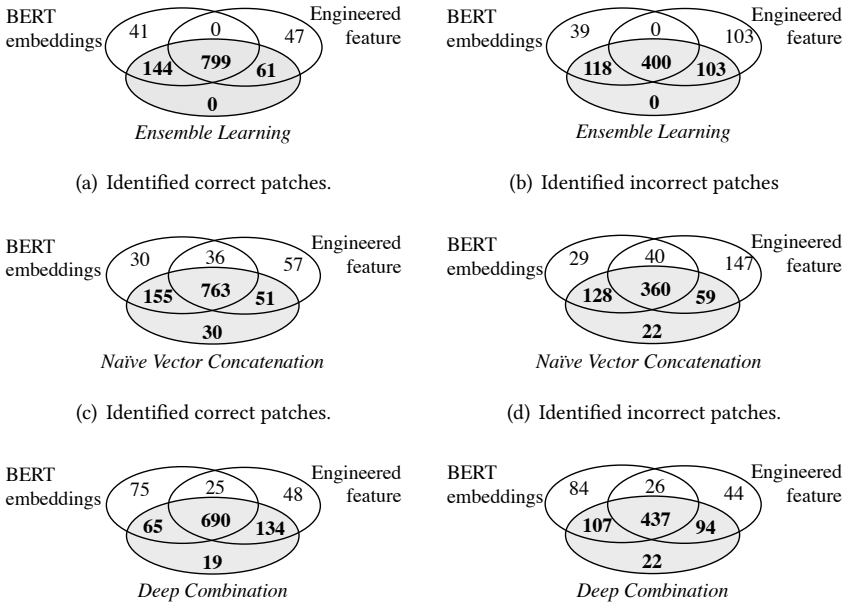
(a) Ensemble learning. (b) Naïve Vector Concatenation. (c) Deep Combination.

Fig. 12. Combination options of features for patch classification in PANTHER.

Table 11. Comparing results of classifying correct patches with combined feature against the single feature.

Tool	Feature	Accuracy	Precision	+Recall	-Recall	F1-measure	AUC
			Random Forest				
LEOPARD	BERT embeddings	0.694	0.683	0.779	0.624	0.755	0.793
	Engineered feature	0.707	0.721	0.775	0.620	0.747	0.769
PANTHER	*Ensemble Learning*	0.745	0.740	**0.837**	**0.629**	0.786	**0.818**
	Naïve Vector Concatenation	0.708	0.693	0.786	0.629	0.766	0.799
			XGBoost				
LEOPARD	BERT embeddings	0.718	0.716	0.821	0.588	0.765	0.803
	Engineered feature	0.705	0.726	0.799	0.596	0.741	0.776
PANTHER	*Ensemble Learning*	0.757	0.754	**0.837**	**0.655**	0.794	**0.822**
	Naïve Vector Concatenation	0.730	0.725	0.833	0.600	0.775	0.811
			DNN				
LEOPARD	BERT embeddings	0.703	0.744	0.713	0.690	0.728	0.767
	Engineered feature	0.698	0.721	0.748	0.634	0.734	0.777
PANTHER	*Deep Combination*	0.730	0.760	**0.757**	**0.696**	0.758	**0.798**

incorrect patches can be filtered out), and AUC (area under the ROC curve, i.e. comprehensive performance of the predictor). Overall, the performance of classifying correct patches is improved after using each of the three combination strategies (except the -Recall of the random forest classifier with the *Naïve Vector Concatenation*) for the learned (BERT) and engineered (ODS) feature. With respect to **+Recall** (i.e., recalling the correct patches), the Random forest and XGBoost based classifier with *Ensemble Learning* achieve the highest value at 83.7%, improving by 1 to 6 percentage points the performance with single feature sets. With respect to **-Recall** (i.e., filtering out the incorrect patches), the best classifier is DNN-based with the *Deep Combination* of features: it achieves the highest recall in correctly excluding 69.6% of the incorrect patches. With respect to **AUC**, the XGBoost-based classifier with the *Ensemble Learning* present the best performance at 82.2%, improving by 2 to 5 percentage points the performance with single feature sets. To sum up, combining the BERT embeddings of patches with their ODS features does improve the performance of identifying patch correctness. Note that the results show that, in general, Ensemble Learning applied to independently trained classifiers yields the highest performance gains. The McNemar's statistical hypothesis test [10] further confirms that the gains are statistically significant for the *Ensemble Learning* and *Deep Combination* while it is not the case for the *Naïve Vector Concatenation*. This suggest that the features (learned and engineered) are from different spaces and are best exploited when applied standalone to model patch correctness, and can complement each other in terms of prediction.



(a) Identified correct patches. (b) Identified incorrect patches

(c) Identified correct patches. (d) Identified incorrect patches.

(e) Identified correct patches. (f) Identified incorrect patches.

Fig. 13. Comparison on the number of patches identified with the combined feature vs. the simple feature.

Figure 13 further highlights the number of (in)correct patches identified based on BERT embeddings, engineered features and the combined features, respectively. Since the "Random forest" learner presents a similar performance with "XGBoost", Figure 13 focuses on the latter.

From a **qualitative point of view**, with the *Ensemble Learning*, more (in)correct patches can be identified than each single feature set (i.e., BERT embeddings or engineered features). However, this combination does not help to identifying patches that were not identified using at least one feature set. In contrast, with *Naïve Vector Concatenation* and the *Deep Combination*, which combine features before classification, we can identify some (in)correct patches that could not be identified using either feature set alone.

From a **quantitative point of view**, the *Naïve Vector Concatenation* helps to identify slightly more correct patches (among those that could not be identified by each feature set alone) than the *Deep Combination*. As for new identified incorrect patches, they achieve the same metrics. Nevertheless, overall, the *Ensemble Learning* method helps to identify more correct patches while the *Deep Combination* helps to identify more incorrect patches.

✍ **RQ-4** ► *Leveraging learned embeddings (BERT) and engineered features (ODS) contributes to improve the performance in predicting patch correctness for APR tools. Merging independently trained classifiers achieves higher performance compared to each separate classifier, but does not lead to the identification of correct/incorrect patches that could not be identified by at least one of the classifier. In contrast, feature combination (i.e., Naïve Vector Concatenation and Deep Combination) before classification training appears to provide more information to discriminate some patches that were not correctly classified based on their learned embeddings or their engineered features alone.* ◄

4.5 [RQ-5: Explanation of Improvements of Combination]

Objective: The experimental results for previous RQs show that ML classifiers built based on learned embeddings, or on engineered features, or on both, yield promising performance in predicting patch correctness. The fact remains, however, that the classifier is a black box model for practitioners. In particular, when leveraging combined feature sets, it may be helpful to investigate the impact of different features on the identification of patch correctness. To that end, we propose to build on Explainable ML techniques to explore how the models are built. In this work, we focus on Shapley Values, which compute the contributions of each feature in a given prediction. Shapley values originate from the field of game theory and have been implemented in the SHAP framework [40], which is widely used in the AI community.

Experimental Design: Our experiments are focused on the classifier yielded with the *Naïve Vector Concatenation* method since it managed to recall more correct patches through combining learned embeddings and engineered features (cf. RQ-3.3 in Section 4.3). We consider the case where the classifier is trained with the XGBoost learning algorithm. Using SHAP values as metric of feature importance, we investigate the top most important features that contribute to the combined model predictions. We further compare those important features against the features that are most contributing when the classifier is trained only with learned embeddings or only with engineered features. Finally, we present three specific patches that identified by different feature sets to observe the contribution of the features to prediction.

Results: Figure 14 illustrates the top-10 most contributing features: a feature named B-*i* refers to the i^{th} feature learned with BERT. Others (e.g., *singleLine* and *codeMove*) refer to engineered features. The appearance of features from learned and engineered feature sets among the most contributing features suggests that both types of features are not only relevant but are also exploited in the yielded classifier.

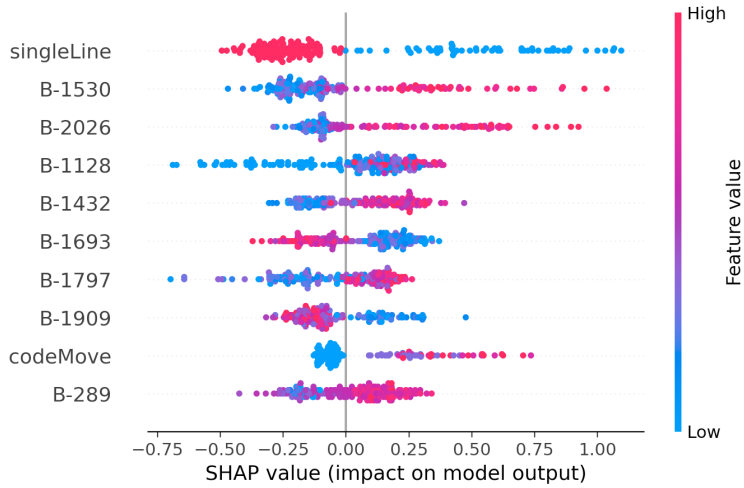


Fig. 14. Top-10 Contributing Features (based on SHAP values) for the Classifier built by combining learned embeddings and engineered features.

Reading a SHAP explanation graph: In a given SHAP graph, each row is a distribution values for a given feature (Y-axis), where each data point is associated to one sample input data (i.e., a patch in our case). The color indicates the feature value, which is normalized: the more red, the higher the value. The X-axis represents the SHAP values, which indicate to what extent a given feature impacted the model output for a given patch. For example, most patches with high value (red) for

feature *singleLine* are located on the left (negative SHAP value), which suggests negative impact of *singleLine* on correctness prediction. It should be noted that, eventually, it is the contributions of different features that will be merged to yield the final prediction for each sample.

In Figure 14, we note that *singleLine* and *codeMove* are the top contributing engineered features among the combined feature sets. As we see from the figure, their red (high value) points and blue (low value) points are clearly separated to two sides, which demonstrates their values have obvious positive or negative effects on the model output. In Figure 15, when leveraging only engineered features, *singleLine* and *codeMove* also have significant contributions and are appearing in the 1st and 4th positions among the top contributing features. This indicates that the engineered features must be high-contributors to the decision (e.g., in terms of information gain) as shown in Figure 15, in order to obtain an efficient combination with learned features. Therefore, in practice we suggest that the research community should focus more on devising few but effective engineered features instead of massive but inefficient features to improve the performance of models.

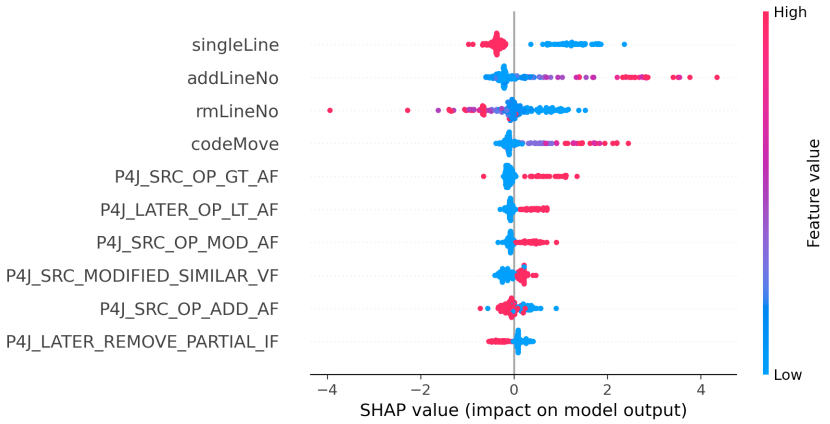
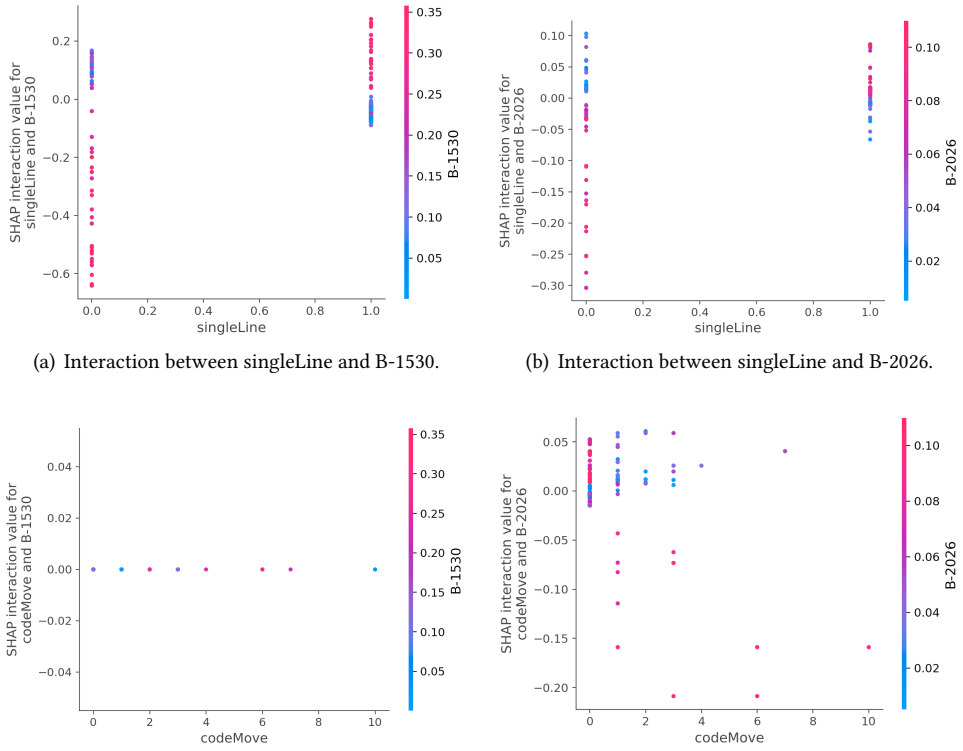


Fig. 15. Top-10 contributing features (based on SHAP values) for the Classifier built only by the engineered features.

Overall, the SHAP explanations suggest that engineered features have an important effect on model prediction (because they appear among the top contributing features) but are complementary to the learned feature set. Indeed, the combination with *Naive Vector Concatenation* enables classifiers to identify correct patches that could not be identified when each feature set was used without the other. Therefore, we conclude that it is the interaction among the features that yields such a performance improvement. We propose to further investigate the interaction among pairs of features (one from the engineered features set and the other from the learned features set).

Figure 16 illustrates the interactions information provided by SHAP among *singleLine*, *codeMove* and *B-1530*. As it can be seen, in Figure 16(a), when the feature value of *singleLine* is 0, higher (redder) feature values of *B-1530* will lead to a more negative SHAP value for *singleLine* (i.e., it has negative impact on patch correctness prediction). In contrast, when the feature value of *singleLine* is 1, the same higher feature values of *B-1530* will tend to draw a positive SHAP value (i.e., positive impact). This example illustrates how learned and engineered features can interact to balance their contributions for the final predictions based on their respective feature values. Figure 16(b) and Figure 16(d) exhibit effective interaction while Figure 16(c) cannot because not enough of the test data are reaching both the two feature nodes in the tree-based boosting classifier. In the same direction, we cannot present the SHAP interaction between *singleLine* and *codeMove*. Overall, Figure 16 provides evidence for the impact of the interaction between learned and engineered features on the model prediction. In contrast, merging classifiers through *Ensemble Learning* does

not allow for features interaction and thus fails to identify patches that were not identified using one feature set. This motivates model trainers to combine different types of features through tree-based classifiers or deep neural networks to obtain efficient deep information for identifying previously-unidentified correct patches.



(a) Interaction between singleLine and B-1530.

(b) Interaction between singleLine and B-2026.

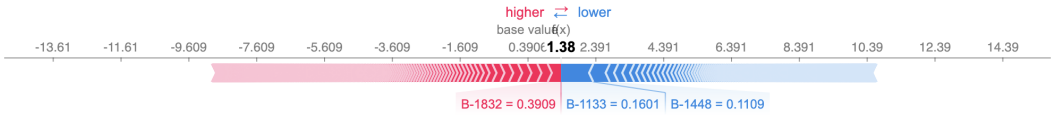
(c) Interaction between codeMove and B-1530.

(d) Interaction between codeMove and B-2026.

Fig. 16. Feature Interaction.

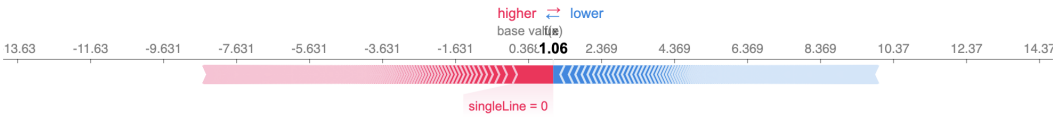
Finally, Figure 17 presents the SHAP analyses of three patches that are exclusively identified by classifiers built based either on learned feature set (a), or on engineered feature set (b), or on combined feature set (c). We note that contributions of each learned feature is small and it is the sum of contributions that lead to a prediction. In contrast, contributions of engineered features are significantly larger for several features. When the sets are combined, engineered features are contributing in the top, their contributions are impactful, while learned features still contribute, each, to a lesser extent. Overall, few engineered features make most of the contributions for good prediction which unsurprisingly imply that the quality and relevance of engineered features are more important than the number of features.

✍ **RQ-5** ► *Thanks to SHAP explanations, we were able to confirm that combining engineered and learned feature sets creates interactions that impact the prediction of classifiers, leading to improved precision and recall in correctness prediction.* ◄



(a) Patch for Closure-57.

(b) Patch for Math-85.



(c) Patch for Math-56.

Fig. 17. SHAP Analysis on Patches.

5 DISCUSSIONS

We enumerate a few insights from our experiments with representation learning models and discuss some threats to validity.

5.1 Insights from the Experimental Setup

Code-oriented embedding models may not yield the best embeddings for training patch correctness classifiers. Our experiments have revealed that the BERT model, which was pre-trained on Wikipedia, is yielding the best recall in the identification of incorrect patches. There are several possible reasons for this situation: BERT implements the deepest neural network and builds on the largest training data. Its performance suggests to researchers that code-oriented embedding models should be trained on large code datasets or fine-tuned on specific target tasks in order to become competitive against BERT. While we were completing the experiments, a pre-trained CodeBERT [13] model has been released. In future work, we will investigate its relevance for producing embeddings that may yield higher performance in patch correctness prediction. In any case, we note that CC2Vec provided the best embeddings for yielding the best recall in identifying correct patches (using similarity thresholds). This finding suggests we use the embedding model built for code changes (e.g., CC2Vec) for the objective of having a high recall in identifying correct patches.

The small sizes of the code fragments lead to similar embeddings. Figure 18 illustrates the different cosine similarity scores that can be obtained for the BERT embeddings of different pairs of short sentences. Although the sentences are semantically (dis)similar, the cosine similarity scores are quite close. This explains why recalling correct patches based on a similarity threshold was a failed attempt ($\sim$ 5% for APR-generated patches for Defects4J+Bears+Bugs.jar bugs). Nevertheless, experimental results demonstrated that deep learned features are relevant for learning to discriminate. Considering the different sizes of code fragments contained in each patch may affect the similarity computation, we suggest that researchers control the size of the code fragments of the patch when investigating the hypothesis in RQ-2 for patch correctness.

Refutation of literature assumption that "patches with fewer changes are more likely to be correct". In RQ-2, we leveraged similarity between buggy code and patched code to filter out incorrect patches. The hypothesis is the more similar they are, the more likely to be correct the patch is. The best performance appears in QuixBugs that only contain bug on one single line. However, regarding

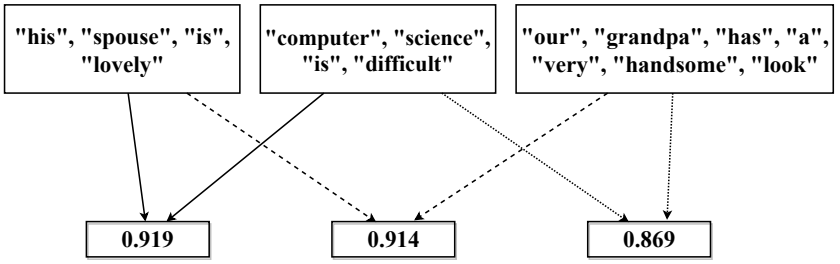


Fig. 18. Close cosine similarity scores with small-sized inputs for BERT embedding model.

Bears, Bugs.jar and Defects4j, while a large number of incorrect patches are filtered out (cf. -Recall in Table 6), correct patches are recalled in low numbers (cf. +Recall in Table 6). Or, -Recall is low while keeping high +Recall. In the RQ-5, we use ground-truth labeled developer's patches and generated patches with balanced numbers for Defects4j to avoid bias. We use SHAP to interpret the impact of feature and find the most important feature is "singleLine". The feature analysis implies that patch with one single line (fewer change) is more likely to be incorrect, which is against the hypothesis. This demonstrates correct code normally require more than one-line change.

5.2 Threats to Validity

Our empirical study carries a number of threats to validity that we have tried to mitigate.

Threats to External Validity. There are a variety of representation learning models in the literature. A threat to validity of our study is that we may have a selection bias by considering only four embedding models. We have mitigated this threat by considering representative models in different scenarios (pre-trained vs retrained, code change specific vs natural language oriented).

Another threat to validity is related to the use of Defects4J data in evaluating the ML classifiers. This choice however was dictated by the data available and the aim to compare against related work.

Threats to Internal Validity. A major threat to internal validity lies in the manual assessment heuristics that we applied to the RepairThemAll-generated dataset. We may have misclassified some patches due to mistakes or conservatism. This threat however holds for all APR work that relies on manual assessment. We mitigate this threat by following clear and reproducible decision criteria, and by further releasing our labelled datasets for the community to review[8]. Besides, we supplement the dataset with developer patches to mainly relieve the imbalance problem of the dataset. This may make the sample distribution of our experiment different from the real APR patches world. This threat however also holds for some current works [66, 71] that focus on patch correctness validation.

Threats to Construct Validity. For our experiment, the considered embedding models are not perfect and they may have been under-trained for the prediction task that we envisioned. For this reason, the results that we have reported are likely an under-estimation of the capability of representation learning models to capture discriminative features for the prediction of patch correctness. Our future studies on representation learning will address this threat by considering different re-training experiments.

[8]see: https://github.com/HaoyeTianCoder/Panther

6 RELATED WORK

Analyzing Patch Correctness: To assess the performance of fixing bugs of repair tools and approaches, checking the correctness of patches is key, but not trivial. However, this task was largely ignored or unconcerned in the community until the analysis study of patch correctness conducted by Qi *et al.* [51]. Thanks to their systematic analysis of the patches reported by three generate-and-validate program repair systems (i.e., GenProg, RSRepair and AE), they shown that the overwhelming majority of the generated patches are not correct but just overfit the test inputs in the test suites of buggy programs. In another study, Smith *et al.* [54] uncover that patches generated with lower coverage test suites overfit more. Actually, these overfitting patches often simply break under-tested functionalities, and some of them even make the "patched" program worse than the un-patched program. Since then, the overfitting issue has been widely studied in the literature. For example, Le *et al.* [25] revisit the overfitting problem in semantics-based APR systems. In [24], they further assess the reliability of authors and automated annotations in assessing patch correctness. They recommend to make publicly available to the community the patch correctness evaluations of the authors. Yang [69] explore the difference between the runtime behavior of programs patched with developer's patches and those by APR-generated plausible patches. They unveil that the majority of the APR-generated plausible patches lead to different runtime behaviors compared to correct patches.

Predicting Patch Correctness: To predict the correctness of patches, one of the first explored research directions relied on the idea of augmenting test inputs, i.e., more tests need to be proposed. Yang *et al.* [70] design a framework to detect overfitting patches. This framework leverages fuzz strategies on existing test cases in order to automatically generate new test inputs. In addition, it leverages additional oracles (i.e., memory-safety oracles) to improve the validation of APR-generated patches. In a contemporary study, Xin and Reiss [65] also explored to generate new test inputs, with the syntactic differences between the buggy code and its patched code, for validating the correctness of APR-generated patches. As complemented by Xiong *et al.* [66], they proposed to assess the patch correctness of APR systems by leveraging the automated generation of new test cases and measuring behavior similarity of the failing tests on buggy and patched programs.

Through an empirical investigation, Yu *et al.* [75] summarized two common overfitting issues: incomplete fixing and regression introduction. To assist alleviating the overfitting issue for synthesis-based APR systems, they further proposed `UnsatGuided` that relies on additional generated test cases to strengthen patch synthesis, and thus reduce the generation of incorrect overfitting patches.

The success of predicting patch correctness using an augmented set of test cases, as it is done in prior work, depends on the quality of the tests. In practice, however, tests with high coverage are often unavailable [71]. To overcome this limitation, our approach does not rely on new test cases, but instead leverages learning techniques to build representation vectors for buggy and patched code of APR-generated patches. Patch correctness prediction is therefore conducted without the constraints in the availability of test cases.

To predict overfitting patches yielded by APR tools, Ye *et al.* [71] propose ODS, an overfitting detection system. ODS first statically extracts 4,199 code features at the AST level from the buggy code and generated patch code of APR-generated patches. Those features are fed into three machine learning algorithms (logistic regression, KNN, and random forest) to learn an ensemble probabilistic model for classifying and ranking potentially overfitting patches. To evaluate the performance of ODS, the authors considered 19,253 training samples and 713 testing samples from the Durieux *et al.* empirical study [11]. With these settings, ODS is capable of detecting 57% of overfitting patches. The ODS approach relates to our study since both leverage machine learning and static features.

However, ODS only relies on manually identified features which may not generalize to other programming languages or even other datasets.

In a recent work, Csuvik *et al.* [8] exploit the textual and structural similarity between the buggy code and the APR-patched code with two representation learning models (BERT [9] and Doc2Vec [23]) by considering three patch code representation (i.e., source code, abstract syntax tree and identifiers). Their results show that the source code representation is likely to be more effective in correct patch identification than the other two representations, and the similarity-based patch validation can filter out incorrect patches for APR tools. However, to assess the performance of the approach, only 64 patches from QuixBugs [72] have been considered (including 14 in-the-lab bugs). This low number of considered patches raises questions about the generalization of the approach for fixing bugs in the wild. Moreover, unlike our study, new representation learning models (code2vec [2] and CC2Vec [15]) dedicated to code representation have not been exploited. In our work, we first improve the evaluation of the approaches in the real-world by designing a 10-group cross validation on a large labeled deduplicated dataset of 2,147 patches. Then, we propose an extension of our previous works on predicting patch correctness by combining engineered features [71] and representation learning [57] (BERT, Doc2Vec, CC2Vec) together and assessing the effectiveness of each and their combination as well as the improvement of the combination. Our study aims to show how the combinations can be carried out to ensure that patches that could not be identified by either set of features are not identifiable by the combined set. More recently, Yan et al. [68] proposed to predict the patch correctness of fixing C program bugs through the transfer learning of execution semantics. Tian et al. [58] explored the relationship between the bug descriptions carried by bug reports and code changes to identify the correctness of patches for the given Java program bugs.

Representation Learning for Program Repair Tasks: In the literature, representation learning techniques have been widely explored to boost program repair tasks. Long and Rinard explored the topic of learning correct code for patch generation [39]. Their approach learns code transformation for three kinds of bugs from their related human-written patches. After mining the most recent 100 bug-fixing commits from each of the 500 most popular Java projects, Soto and Le Goues [55] have built a probabilistic model to predict bug fixes for program repair. To identify stable Linux patches, Hoang *et al.* [16] proposed a hierarchical deep learning-based method with features extracted from both commit messages and commit code. Liu *et al.* [31] and Bader *et al.* [3] proposed to learn recurring fix patterns from human-written patches and suggest fixes. Our paper does not propose a new automated patch generation approach. Instead, we fill a gap in the literature by proposing a comprehensive assessment of the effectiveness of different representation learning models on predicting the correctness of patches generated by program repair tools.

7 CONCLUSION

In this paper, we investigated the feasibility of statically predicting patch correctness by leveraging representation learning models and supervised learning algorithms. The objective is to provide insights for the APR research community towards improving the quality of repair candidates generated by APR tools. To that end, we, first investigated the use of different distributed representation learning to capture the similarity/dissimilarity between buggy and patched code fragments. These experiments gave similarity scores that substantially differ for across embedding models such as BERT, Doc2Vec, code2vec and CC2Vec. Building on these results and in order to guide the exploitation of code embeddings in program repair pipelines, we investigated in subsequent experiments the selection of cut-off similarity scores to decide which APR-generated patches are likely incorrect. We then implemented a patch correctness predicting framework, Leopard, to

investigate the discriminative power of the deep learned features by training machine learning classifiers to predict correct Patches. Decision Trees, Logistic Regression, Naïve Bayes, Random Forest, XGBoost, and DNN are tried with code embeddings from BERT, Doc2Vec and CC2Vec. With BERT embeddings, Leopard (with XGBoost) yielded very promising performance on patch correctness prediction with metrics like Recall at 82.1% and F-Measure at 76.5%, Leopard (with DNN) achieved the highest score with the metric Precision at 0.744 on a labeled deduplicated dataset of 2,147 patches. We further showed that the performance of these models on learned embedding features is promising when comparing against the state of the art (PATCH-SIM [66]), which uses dynamic execution traces. We further implemented Panther (an upgraded version of Leopard) to explore the combination of the learning embeddings and the engineered features to improve the performance on identifying patch correctness with more accurate classification. Finally, leveraging SHAP, we analyzed the cause of prediction behind features and classifiers to help aware the essence of identifying patch correctness. Since our approach is able to swiftly predict patch correctness, future work should investigate how to incorporate it with APR tools to explore large patch space more efficiently.

Availability. All artifacts of this study are available in the following public repository:

https://github.com/HaoyeTianCoder/Panther

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* 51, 4 (2018), 81:1–81:37. https://doi.org/10.1145/3212695

[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), 40:1–40:29. https://doi.org/10.1145/3290353

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 159:1–159:27. https://doi.org/10.1145/3360585

[4] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 306–317. https://doi.org/10.1145/2635868.2635898

[5] Junjie Chen, Alastair F. Donaldson, Andreas Zeller, and Hongyu Zhang. 2017. Testing and Verification of Compilers (Dagstuhl Seminar 17502). *Dagstuhl Reports* 7, 12 (2017), 50–65. https://doi.org/10.4230/DagRep.7.12.50

[6] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.

[7] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding Java Classes with code2vec: Improvements from Variable Obfuscation. In *Proceedings of the 17th Mining Software Repositories*. ACM.

[8] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *Proceedings of the 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 18–25. https://doi.org/10.1109/IBF50092.2020.9034714

[9] 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*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[10] Thomas G Dietterich. 1998. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation* 10, 7 (1998), 1895–1923.

[11] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[12] 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

[13] 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). https://arxiv.org/abs/2002.08155

[14] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[15] Thong Hoang, Hong Jin Kang, Julia Lawall, and David Lo. 2020. CC2Vec: Distributed Representations of Code Changes. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 518–529. https://doi.org/10.1145/3377811.3380361

[16] Thong Hoang, Julia Lawall, Yuan Tian, Richard Jayadi Oentaryo, and David Lo. 2019. PatchNet: Hierarchical Deep Learning-Based Stable Patch Identification for the Linux Kernel. *CoRR* abs/1911.03576 (2019). http://arxiv.org/abs/1911.03576

[17] 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

[18] 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

[19] 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 23rd International Symposium on Software Testing and Analysis*. ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[20] Rafael-Michael Karampatsis and Charles A. Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *Proceedings of the 17th Mining Software Repositories*. IEEE. http://arxiv.org/abs/1905.13334

[21] 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

[22] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug Report driven Program Repair. In *Proceedings of the 27the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 314–325. https://doi.org/10.1145/3338906.3338935

[23] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning*. JMLR.org, 1188–1196. http://proceedings.mlr.press/v32/le14.html

[24] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 524–535. https://doi.org/10.1109/ICSE.2019.00064

[25] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033. https://doi.org/10.1007/s10664-017-9577-2

[26] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[27] 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

[28] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[29] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 55–56. https:

//doi.org/10.1145/3135932.3135941
[30] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 1–12. https://doi.org/10.1109/ICSE.2019.00019
[31] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. 2018. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2884955
[32] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution*. IEEE, 275–286. https://doi.org/10.1109/ICSME.2018.00037
[33] 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
[34] 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
[35] 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
[36] 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
[37] 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
[38] 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 42nd International Conference on Software Engineering*. ACM, 625–627. https://doi.org/10.1145/3377811.3380338
[39] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Vol. 51. ACM, 298–312. https://doi.org/10.1145/2837614.2837617
[40] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf
[41] Fernanda Madeiral, Thomas Durieux, Victor Sobreira, and Marcelo Maia. 2018. Towards an automated approach for bug fix pattern detection.
[42] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 468–478. https://doi.org/10.1109/SANER.2019.8667991
[43] Henry B Mann and Donald R. Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. https://doi.org/10.1214/aoms/1177730491
[44] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205. https://doi.org/10.1007/s10664-013-9282-8
[45] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[46] Martin Monperrus. 2018. Automatic software repair: A bibliography. *Comput. Surveys* 51, 1 (2018), 17:1–17:24. https://doi.org/10.1145/3105906
[47] Martin Monperrus. 2018. The living review on automated program repair. In *HAL/archives-ouvertes. fr, Technical Report*.
[48] Samuel Ndichu, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. 2019. A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors. *Applied Soft Computing* 84 (2019). https://doi.org/10.1016/j.asoc.2019.105721

[49] Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2022. MetaTPTrans: A Meta Learning Approach for Multilingual Code Representation Learning. *arXiv preprint arXiv:2206.06460* (2022).

[50] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 254–265. https://doi.org/10.1145/2568225.2568254

[51] 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 24th International Symposium on Software Testing and Analysis*. ACM, 24–36. https://doi.org/10.1145/2771783.2771791

[52] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*. ACM, 10–13. https://doi.org/10.1145/3196398.3196473

[53] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 13–24. https://doi.org/10.1109/ICSE.2019.00020

[54] 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

[55] Mauricio Soto and Claire Le Goues. 2018. Using a probabilistic model to predict bug fixes. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 221–231. https://doi.org/10.1109/SANER.2018.8330211

[56] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2022. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. *ACM Transactions on Software Engineering and Methodology* (2022).

[57] 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.

[58] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2022. Is this Change the Answer to that Problem? Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE.

[59] 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*. ACM.

[60] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 3034–3040. https://doi.org/10.24963/ijcai.2017/423

[61] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 356–366. https://doi.org/10.1109/ASE.2013.6693094

[62] 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

[63] 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

[64] F. Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[65] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 226–236. https://doi.org/10.1145/3092703.3092718

[66] 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*. ACM, 789–799. https://doi.org/10.1145/3183519.3183540

[67] 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

[68] Dapeng Yan, Kui Liu, Yuqing Niu, Li Li, Zhe Liu, Zhiming Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2022. Crex: Predicting patch correctness in automated repair of C programs through transfer learning of execution semantics. *Information and Software Technology* 152 (2022), 107043. https://doi.org/10.1016/j.infsof.2022.107043

[69] Bo Yang and Jinqiu Yang. 2020. Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level. In *Proceedings of the 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 1–8. https://doi.org/10.1109/IBF50092.2020.9034821

[70] 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. https://doi.org/10.1145/3106237.3106274

[71] 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).

[72] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. In *Proceedings of the 1st International Workshop on Intelligent Bug Fixing*. IEEE, 1–10. https://doi.org/10.1109/IBF.2019.8665475

[73] He Ye, Matias Martinez, and Martin Monperrus. 2019. Automated patch assessment for program repair at scale. *arXiv preprint arXiv:1909.13694* (2019).

[74] 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 AAAI Conference on Artificial Intelligence*. AAAI, 1145–1152. https://doi.org/10.1609/aaai.v34i01.5466

[75] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering* 24, 1 (2019), 33–67. https://doi.org/10.1007/s10664-018-9619-4

[76] 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.

[77] 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. https://doi.org/10.1109/ASE.2019.00137