APPT: Boosting Automated Patch Correctness Prediction via Fine-tuning Pre-trained Models

Quanjun Zhang, Chunrong Fang, Weisong Sun, Yan Liu, Tieke He, Xiaodong Hao, Zhenyu Chen

Abstract—Automated program repair (APR) aims to fix software bugs automatically without human debugging efforts and plays a crucial role in software development and maintenance. Despite the recent significant progress in the number of fixed bugs, APR is still challenged by a long-standing overfitting problem (i.e., the generated patch is plausible but overfitting). Various techniques have thus been proposed to address the overfitting problem. Recently, researchers have employed BERT to extract code features, which are then used to train a classifier for patch correctness prediction, indicating the potential of such pre-trained models in reasoning about patch correctness. However, BERT is restricted to feature extraction for classifier training without benefiting from the training process, potentially generating sub-optimal vector representations for patched code snippets. In this paper, we propose APPT, a pre-trained model-based automated patch correctness assessment technique by both pre-training and fine-tuning. APPT adopts a pre-trained model as the encoder stack, followed by an LSTM stack and a deep learning classifier. More importantly, the pre-trained model is fine-tuned in conjunction with other components as a whole pipeline to fully adapt it specifically for reasoning about patch correctness. Although our idea is general and can be built on various existing pre-trained models, we have implemented APPT based on the BERT model. We conduct an extensive experiment on 1,183 Defects4J patches and the experimental results show that APPT achieves prediction accuracy of 79.7% and recall of 83.2%, outperforming the state-of-the-art technique CACHE by 4.3% and 6.7%. Our additional investigation on 49,694 real-world patches shows that APPT achieves the optimum performance (exceeding 99% in five common metrics for assessing patch classification techniques) compared with existing representation learning techniques. We further investigate the impact of each component and find that they all positively contribute to APPT, e.g., the fine-tuning process and the LSTM stack increase F1-score by 10.22% and 4.11%, respectively. We also prove that adopting advanced pre-trained models can further provide substantial advancement (e.g., GraphCodeBERT-based APPT improves BERT-based APPT by 2.8% and 3.3% in precision and AUC, respectively), highlighting the generalizability of APPT. Overall, our study highlights the promising future of fine-tuning pre-trained models to assess patch correctness and reduce the manual inspection effort of debugging experts when deploying APR tools in practice.

Index Terms—Automated Program Repair, Patch Correctness, Pre-trained Model

✦

1 INTRODUCTION

Software bugs are inevitable in modern software systems and result in fatal consequences, such as costing trillions of dollars in financial loss and affecting billions of people around the world [1], [2]. It is incredibly time-consuming and labor-intensive for developers to fix such bugs due to the increasing size and complexity of modern software systems [3], [4]. Automated program repair (APR) aims to fix revealed software bugs without human intervention automatically and has attracted massive attention from both academia and industry in the past decades [5]–[7]. Despite an emerging research area, a variety of APR techniques have been proposed and continuously achieved promising results in terms of the number of fixed bugs in the literature [8], [9].

However, it is fundamentally difficult to achieve high precision for generated patches due to the weak program specifications [10]. Existing APR techniques usually leverage the developer-written test cases as the criteria to assess the correctness of the generated patches. In fact, a generated patch passing the available test cases may not generalize to other potential test cases, leading to a long-standing challenge of APR (i.e., the overfitting issue) [10], [11]. For example, when a bug is detected in functionality, a patch can be simply generated by deleting the functionality and the available test cases usually fail to exercise the deleted functionality [12]. In this case, developers need to consume tremendous time and effort to filter the overfitting patches, resulting in a negative debugging performance when APR techniques are applied in practice [13]–[15].

Thus, various automated patch correctness assessment (APCA) techniques have been proposed to determine whether a generated patch is indeed correct or not [16]. According to extracted features, the traditional APCA techniques can be categorized into two groups: static and dynamic ones [17]. Static techniques tend to analyze the code changed patterns or code similarity based on the syntactic and semantic features. For example, Tan et al. [18] define a set of generic forbidden transformations (e.g., the above-mentioned functionality deleting) for the buggy program. In contrast, dynamic techniques usually execute the plausible patches against extra test cases generated by automated test generation tools (e.g., Evosuite [19] and Randoop [20]). For example, Xiong et al. [21] generate new test cases and determine patch correctness based on the behavior similarity of the test case executions. However, the static techniques may suffer from prediction precision problems, while it is

---

- *Quanjun Zhang, Chunrong Fang, Weisong Sun, Yan Liu, Tieke He, Xiaodong Hao and Zhenyu Chen are with the State Key Laboratory for Novel Software Technology, Nanjing University, China.*
  E-mail: quanjun.zhang@smail.nju.edu.cn, fangchunrong@nju.edu.cn, weisongsun@smail.nju.edu.cn, MF21320104@smail.nju.edu.cn, hetieke@nju.edu.cn, MF21320054@smail.nju.edu.cn, zychen@nju.edu.cn
- *Chunrong Fang, Tieke He, and Zhenyu Chen are the corresponding authors.*

pretty time-consuming for dynamic techniques to generate additional test cases and execute all patched programs [17].

Recently, inspired by large-scale patch benchmarks being released [8], [9], some learning-based APCA techniques have been proposed to assess patch correctness by embedding buggy and patched code snippets [16], [22], [23]. For example, He et al. [22] extract hand-crafted features from Java programs statically and train a probabilistic model (i.e., ODS) to perform patch correctness prediction. Despite appealing, the construction of carefully hand-crafted features requires professional knowledge in the domain and it is challenging to generalize such features to other scenarios (e.g., different programming languages) [24]. Instead of hand-crafted features, Tian et al. [16] investigate the feasibility of embedding features to build predictive models. Among investigated embedding models, the pre-trained BERT achieves optimal results, demonstrating the potential of such pre-trained models in reasoning about patch correctness. However, BERT is only responsible for extracting features to train the classifier and does not benefit from the training process. Besides, such pre-trained models are usually trained to derive generic knowledge by self-supervised training on various types of corpora (e.g., natural language), thus may generate sub-optimal vector representations for patched code snippets, limiting prediction performance.

This Paper. In this work, we propose, *APPT*, an *A*utomated *P*re-trained model-based *P*atch correc*T*ness assessment technique, which employs both the pre-training and fine-tuning to address the above limitation of prior work. We first adopt the large pre-trained model as the encoder stack to extract code representations. We then employ bidirectional LSTM layers to capture rich dependency information between the buggy and patched code snippets. Finally, we build a deep learning classifier to predict whether the patch is overfitting or not. Unlike ODS, APPT treats only the source code tokens as the input and automatically extracts code features using a well-trained encoder stack, getting rid of the need for manually-designed features [22]. Besides, different from Tian et al. [16] only training the classifiers, APPT is able to further fine-tune the BERT model to obtain the optimal embedding vectors for patch correctness. This domain adaptation is expected to make the model's representations more relevant to distinguishing correct from overfitting patches. Although APPT is conceptually general and can be built on various pre-trained models, we have implemented APPT as a practical APCA tool based on the BERT model. Our experimental results on 1,183 Defects4J patches indicate that APPT improves the state-of-the-art technique CACHE by 4.36% accuracy, 1.3% precision, 6.7% recall, 3.8% F1-score and 2.2% AUC. We conduct an additional investigation on 49,694 real-world patches from five different patch benchmarks and the results show that APPT exceeds 99% in accuracy, precision, recall, F1-score and AUC metrics, outperforming the existing representation learning techniques. Our ablation study demonstrates that the components in APPT all positively contribute to APPT. For example, the improvement achieved by the fine-tuning process reaches 1.82%~13.16% for all metrics. We adopt different pre-trained models to further investigate the generalization ability of APPT. The results demonstrate that APPT with advanced pre-trained models can enhance the
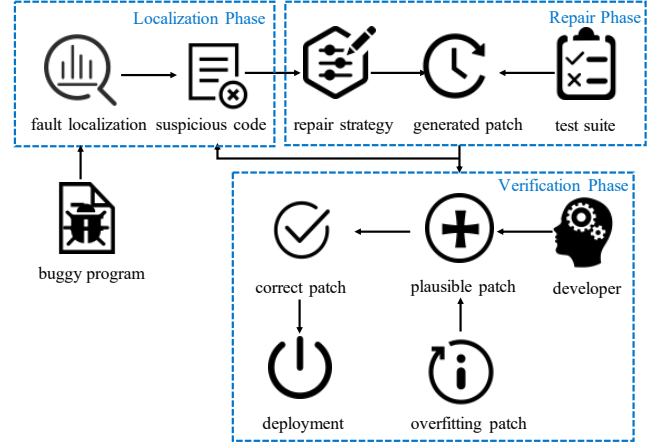

Fig. 1: Overview of APR

prediction performance. For example, precision and AUC of APPT can be improved by 2.8% and 3.3% when equipped with GraphCodeBERT, which are 4.2% and 5.4% higher than the state-of-the-art technique CACHE.

To sum up, we make the following major contributions:

- **Prediction Pipeline.** This paper introduces a prediction pipeline for patch correctness assessment, leveraging large pre-trained models through a process of pre-training followed by fine-tuning. Compared with existing representation learning techniques, the pre-trained model is further fine-tuned with other components in APPT architecture to obtain optimal embedding vectors for reasoning about patch correctness.
- **Novel Technique.** We propose APPT, a BERT-based APCA technique that leverages the pre-training and classifier to predict patch correctness. To the best of our knowledge, we are the first to exploit fine-tuning the pre-trained model for assessing patch correctness.
- **Extensive Study.** We conduct various empirical studies to investigate and evaluate APPT on diverse patch benchmarks. The results show that APPT achieves significantly better overall performance than existing learning-based and traditional APCA techniques.
- **Available Artifacts.** We release the relevant materials (including source code, patches and results) used in the experiments for replication and future research[1].

2 BACKGROUND

2.1 Automated Program Repair

APR techniques' primary objective is to identify and fix program bugs automatically. Fig. 1 illustrates the workflow of the typical APR technique, which is usually composed of three steps: (1) the localization phase utilizes off-the-shelf fault localization techniques to recognize the suspicious code elements (e.g., statements or methods) [25], [26]; (2) the repair phase then modifies these elements based on a set of transformation rules to generate various new program variants, also called candidate patches; (3) the verification phase adopts the original test cases as the oracle to check whether

---

1. All artifacts relevant to this work can be found at https://github.com/iSEngLab/APPT, accessed March 2023.

candidate patches execute as expected or not. Specifically, a candidate patch passing the original test cases is called a *plausible* patch. A plausible patch that is semantically equivalent to the developer patch denotes a *correct* patch; otherwise, it is an *overfitting* patch.

It is fundamentally challenging to ensure the correctness of the plausible patches due to the weak specification of the program behavior in practice. Existing studies have demonstrated that manually identifying the overfitting patches is time-consuming and may harm the debugging performance of developers [13], [27]. Thus, various techniques have been proposed to validate patch correctness automatically. According to whether the dynamic execution or machine learning is required [17], we categorize them into three main categories: static-based techniques, dynamic-based techniques and learning-based techniques.

• *Static-based APCA techniques.* These techniques aim to prioritize correct patches over overfitting ones by static code features, such as code-deleting program transformations.

• *Dynamic-based APCA techniques.* These techniques aim to filter out overfitting patches by executing extra test cases, which are generated based on fixed or patched programs. According to whether the correct patches are required, these techniques can be further categorized into *dynamic with oracle-based ones* and *dynamic without oracle-based ones*.

• *Learning-based APCA techniques.* These techniques aim to predict the correctness of plausible patches enhanced by machine learning techniques. They usually extract the manually-designed code features and then adopt a classifier to perform patch prediction [22]. Some techniques are proposed to adopt code embedding techniques to extract code features automatically [28], which are also denoted as *representation learning-based APCA techniques*.

Recently, an increasing number of research efforts have attempted to use machine learning techniques to learn from existing patch benchmarks for predicting potential patch correctness, achieving promising results. In this work, we adopt the large pre-trained model (i.e., BERT) to encode plausible patches and train a deep learning classifier to predict patch correctness. Compared to existing techniques, our paper is the first work to predict patch correctness by pre-training and fine-tuning the pre-trained model.

2.2 Pre-trained Model

Recently, Pre-trained models (e.g., BERT) have significantly improved performance across a wide range of natural language processing (NLP) tasks, such as machine translation and text classification [29]–[31]. Typically, the models are pre-trained to derive generic language representations by self-supervised training on large-scale unlabeled data and then are transferred to benefit multiple downstream tasks by fine-tuning on limited data annotation.

Existing pre-trained models usually adopt the encoder-decoder architectures, where an encoder encodes an input sequence as a fixed-length vector representation, and a decoder generates an output sequence based on the input representation. Encoder-only models (e.g., BERT [29]) usually pre-train a bidirectional transformer in which each token can attend to each other. Encoder-only models are good at understanding tasks (e.g., code search), but their bidirectionality nature requires an additional decoder for generation tasks, where this decoder initializes from scratch and cannot benefit from the pre-training tasks. Decoder-only models (e.g., GPT [30]) are pre-trained using unidirectional language modeling that only allows tokens to attend to the previous tokens and itself to predict the next token. Decoder-only models are good at auto-regressive tasks like code completion, but the unidirectional framework is sub-optimal for understanding tasks. Encoder-decoder models (e.g., T5 [31]) often make use of denoising pre-training objectives that corrupt the source input and require the decoder to recover them. Compared to encoder-only and decoder-only models that favor understanding and auto-regressive tasks, encoder-decoder models can support generation tasks like code summarization. In this work, we treat the patch correctness assessment as a binary classification task, and we consider encoder-only models to get embeddings of code snippets according to existing work [32]. Our focus is to investigate the potential of transferring the rich general knowledge acquired during the pre-training phase to the downstream task (i.e., reasoning about patch correctness) via fine-tuning.

Inspired by the success of pre-trained models in NLP, many recent attempts have been adopted to boost numerous code-related tasks (e.g., code summarization and code search) with pre-trained models (e.g., GraphCodeBERT) [33]. Despite the promising results, little work aims to explore the capabilities of pre-trained models in supporting patch correctness assessment. In this work, BERT is selected to exploit pre-trained models for automated patch correctness assessment, as it has been widely adopted in various code-related tasks and is quite effective for classification tasks [34], [35]. Two advanced BERT-style models (i.e., CodeBERT and GraphCodeBERT) are also selected to investigate the generalization ability of APPT.

3 APPROACH

Fig. 2 presents the overall framework of our approach. Generally, APPT accepts a buggy program and a plausible patch that passes the available test cases as inputs. APPT extracts the buggy code snippet and its corresponding patched code snippet, and truncates the code tokens for embedding. APPT then uses the pre-trained BERT model for embedding the truncated tokens. After obtaining the representations for the buggy and patched code snippets, APPT uses four pre-defined functions for integrating the representations. Finally, APPT adopts a deep learning classifier to return the final result (i.e., correct or overfitting).

3.1 Code Extraction

Given a buggy program, existing APR tools may return a plausible patch p (if it exists) that passes all available test cases. *Code extraction phase* aims to take the returned patch and the buggy program as the inputs, and output the corresponding buggy and patched code tokens (shown in Fig. 2(a)).

Specifically, we get the buggy and patched code snippets (i.e., C_b and C_p) by parsing the patch file. Firstly, we select removed and added lines as the buggy and patched lines,
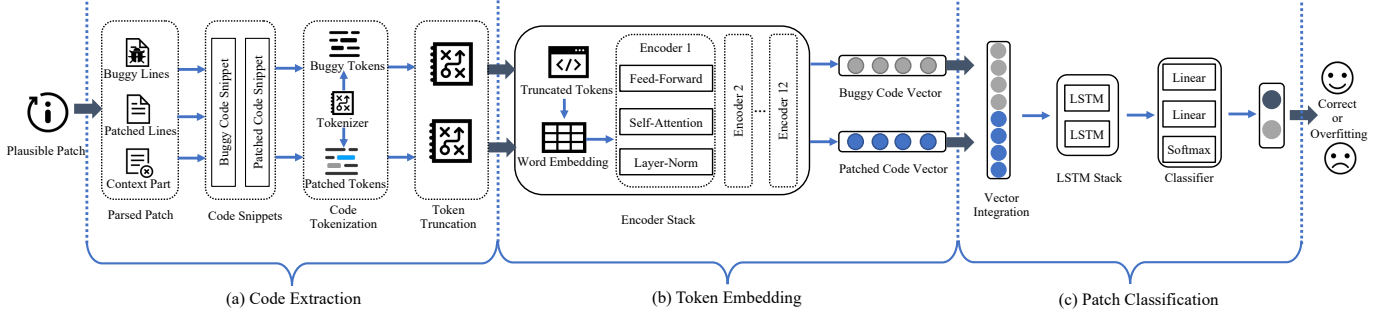
Fig. 2: Overview of APPT

marked with "+" and "-", respectively. Secondly, to keep the context information about the plausible patch, we keep unchanged lines (i.e., without "+" and "-" in the beginning) as part of each code snippet. Finally, the buggy (or patched) code snippet is made up of the buggy (patched) lines and common context part.

We treat the buggy (or patched) code snippet as sequences of tokens and utilize a subword tokenization method to address the out-of-vocabulary problem by breaking down identifiers into their subtokens [36] when tokenizing the code snippet. In this work, we keep the original tokenization vocabulary instead of building a new vocabulary using the byte-pair-encoding algorithm, such that APPT can inherit the natural language understanding ability and start learning prediction from a good initial point.

After the buggy (or patched) code tokens are extracted, we attempt to take them as inputs into the token embedding phase. However, pre-trained models are usually limited to a particular input length. For example, BERT can only take input sequences up to 512 tokens in length. We further truncate the inputs whose length is longer than 512 after tokenization. In particular, we first parse the given buggy code snippet C_b (and the patched one T_p) into individual code tokens T_b (and T_p). We then check whether the length of the code snippet C_b (and T_p) exceeds 512 tokens and retain only the first 512 tokens in T_b (and T_p) to represent the code snippet. Finally, the buggy and patched code tokens (i.e., T_b and T_p) are extracted and truncated based on C_b and T_p to fit the maximum length limit of BERT.

3.2 Token Embedding

Token Embedding phase takes the buggy (or patched) code tokens (i.e., T_b or T_p) as input and embeds it into the buggy (or patched) vector (i.e., E_b or E_p) as output (shown in Fig. 2(b)). APPT implements a stack of twelve layers of encoder blocks to extract the hidden state of the code snippet. Each encoder block consists of three components. The first part is a multi-head self-attention layer to learn long-range dependencies in the input code tokens. The second part is a simple, position-wise fully connected feed-forward neural network, which can linearly transform the token embedding for better feature extraction. The third part is a residual connection around each component, followed by a layer normalization to ensure the stability of code token embeddings distribution.

In particular, the self-attention mechanism computes the representation of each code token by considering the

position relationship between the code tokens. It mainly relies on three main vectors, query Q, key K, and value V, by mapping a query and a set of key-value pairs to an output vector. We employ a scaled dot-product self-attention to calculate the attention scores of each token by taking the dot product between all of the query vectors and key vectors. The attention scores are then normalized to probabilities using the softmax function to get the attention weights. Finally, the value vectors can be updated by taking a dot product between the value vectors and the attention weight vectors. The self-attention operation is computed using three matrices Q, K and V as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad (1)$$

To capture richer semantic meanings of the input code tokens, we further use a multi-head mechanism to realize the self-attention, which allows the model to jointly attend the information from different code representation subspaces at different positions. For d-dimension Q, K, and V, we split those vectors into h heads where each head has d/h-dimension. After all of the self-attention operations, each head will then be concatenated back again to feed into a fully-connected feed-forward neural network including two linear transformations with a ReLU activation in between. The multi-head mechanism can be summarized by the following equation:

$$\text{MultiHead}(Q, K, V) = \text{Concat}\left(\text{head}_1, \ldots, \text{head}_h\right)W^O \qquad (2)$$

where $head_i = Attention(QW_i^Q, KW_i^Q, VW_i^Q)$ and W^O is used to linearly project to the expected dimension after concatenation. Therefore, the encoder stack can take an input code snippet and output a real-valued vector for each code token within the code snippet based on the context. Besides, to obtain optimal embedding vectors in the APCA domain, the encoder stack is further fine-tuned with other components as a whole prediction pipeline.

3.3 Patch Classification

After the embedding vectors of the buggy and patched code snippets (i.e., E_b and E_p) are extracted by the encoder stack, *patch classification phase* first integrates the two vectors into a single input vector (i.e., E_{con}) and then adopts a deep learning classifier to predict the patch correctness automatically (shown in Fig. 2(c)).

3.3.1 Representations Integration

Given two vectors E_b and E_p with n dimensions representing the buggy and patched code snippets, respectively, we integrate the two vectors into one code-changed vector for patch classification. There exist different approaches to integrate E_b and E_p with the aim to characterize their differences from diverse aspects, such as a vector-wise concatenation operation E_{con}, element-wise addition operation E_{add}, element-wise subtraction operation E_{sub}, Hadamard product E_{pro}.

(1) **E_{con}** is a concatenation operation between E_b and E_p on vector-wise level with $2n$ dimension (i.e., $E_{con} = E_b \bigoplus E_p$).

(2) **E_{add}** is an addition operation between E_b and E_p on element-wise level with n dimensions (i.e., $E_{add} = E_b + E_p$).

(3) **E_{sub}** is a subtraction operation between E_b and E_p on element-wise level with n dimensions (i.e., $E_{sub} = E_b - E_p$).

(4) **E_{pro}** is a Hadamard product operation between E_b and E_p on element-wise level with n dimensions (i.e., $E_{sub} = E_b \odot E_p$).

The concatenation operation is proven to be simple yet effective in previous work [24]. Besides, it does not lose the original E_b and E_p vector values, which allows the following LSTM stack to further extract deep relationships between them. Thus, APPT employs the concatenation operation as its default vector integration method. In principle, APPT can also leverage other approaches (e.g., addition and subtraction) for vector integration. We would further investigate the impacts of different vector integration approaches in the detailed experiments.

3.3.2 LSTM Stack

After the embedding vector (e.g., E_{con}) of the changed code tokens is extracted, APPT aims to determine the given patch's correctness based on a deep learning classifier. To extract more hidden code change features, we further feed the code changed vector into a Long Short-Term Memory (LSTM) stack. The LSTM stack has two bidirectional LSTM layers, the output of which is a new state generated by concatenating the hidden states from both directions at a time. LSTM is a specialized recurrent neural network (RNN) for modeling long-term dependencies of sequences. A common LSTM gate unit is composed of a cell, an input gate, an output gate and a forget gate. Thanks to the gated mechanism, LSTM is well-suited to extract the contextual semantic features containing token sequential dependencies and has been widely used in various kinds of tasks, such as vulnerability detection [37], fault localization [38], and automated program repair [39]. Considering that patches may contain changes that are non-adjacent but related, the LSTM stack is adept at capturing these long-range dependencies within the data, which could be instrumental in assessing the correctness of a patch.

In APPT, the LSTM stack computes a mapping from an input code changed vector $x = (x_1, ..., x_T)$ (e.g., E_{con}) to an output vector $z = (z_1, ..., z_T)$ by calculating the network gate unit activations. We implement the gated mechanism by leveraging the input gates and forget gates to control the propagation of cell states. Specifically, when updating the cell state, the input gates decide what new information from the current input to be included in the cell states (i.e., Equation 3), and forget gates decide what information to be excluded from the cell states (i.e., Equation 4). Based on new and forgetting information, cell states as the memory of the LSTM unit can be updated (i.e., Equation 5). The output gate then determines the value for the next hidden state by point-wise multiplication of the output gate (i.e., Equation 6). Finally, the value of the current cell state passed through tanh function (i.e., Equation 7), by which the output of LSTM stack is calculated (i.e., Equation 8).

$$i_t = \text{sigmoid}\left(W_{ix}x_t + W_{ih}h_{t-1} + b_i\right) \tag{3}$$

$$f_t = \text{sigmoid}\left(W_{fx}x_t + W_{fh}h_{t-1} + b_f\right) \tag{4}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh\left(W_{gx}x_t + W_{gh}h_{t-1} + b_g\right) \tag{5}$$

$$o_t = \text{sigmoid}\left(W_{ox}x_t + W_{oh}h_{t-1} + b_o\right) \tag{6}$$

$$h_t = o_t \odot \tanh\left(c_t\right) \tag{7}$$

$$z_t = W_{zh}h_t + b_z \tag{8}$$

where the W terms denote weight matrices (e.g., W_{ix} is the matrix of weights from the input gate to the input), the b terms denote bias vectors (e.g., b_i is the input gate bias vector) and $\odot$ denotes element-wise multiplication of the vectors.

3.3.3 Classifier

After the computation of all LSTM iterations, the embedding vectors of changed code tokens are further fed to a designed deep learning classifier to predict the patch correctness. The classifier is composed of two fully connected layers followed by a binary predictor. In APPT, we apply a standard softmax function to obtain the probability distribution over correctness. A patch is labeled as correct if its probability of being correct is larger than that of being incorrect; otherwise, it is considered overfitting.

In particular, for patch p, z denotes its output of the last iteration in the LSTM stack, which is further linearly transformed into a real number as Equation 9, where $W \in \mathbb{R}^{d \times 1}$, $b \in \mathbb{R}$, and n denotes the number of class (i.e., correct and overfitting). We then leverage softmax function to normalize the output of patch p as Equation 10, where s denotes the correct or overfitting probability of patch p predicted by the model. A patch is considered correct if its probability of being correct is larger than that of being incorrect (i.e., larger than 0.5); otherwise, it is considered overfitting.

$$y_i = W z_i + b_i \quad \forall i \in 1 \ldots n \tag{9}$$

$$s\left(y_i\right) = \frac{\exp\{y_i\}}{\sum_{i=1}^{n} \exp\{y_j\}} \tag{10}$$

3.4 Training

To train the network, we calculate the loss to update the neural weights based on its predicted result and ground truth. We use the cross-entropy loss, which has been widely used in some classification tasks and patch prediction studies [28], [40]. In particular, $g_i \in \{0, 1\}$ denotes whether the i-th patch is correct or overfitting. The cross-entropy loss compares a target g_i with a prediction s in a logarithmic and hence exponential fashion. The objective function is computed in Equation 11, which is minimized constantly in the training to update the parameters in our model. It should be noted that, different from Tian et al. [16] only training the classifier, the whole architecture (i.e., the encoder stack, the LSTM stack, and the clasifier) in our APPT is optimized during training.

$$L = \sum_i -[g_i \cdot \log(s) + (1 - g_i) \cdot \log(1 - s)] \qquad (11)$$

We employ the dropout technique to improve the robustness of APPT and the Adam approach [41] to optimize the objective function.

4 EXPERIMENT

4.1 Research Questions

APPT is designed to predict patch correctness among a mass of plausible patches automatically. To this end, we explore the following research questions (RQ):

RQ1 (Effectiveness): How does APPT perform compared with existing state-of-the-art APCA techniques?

　RQ1.1: How does APPT perform compared with existing state-of-the-art representation learning-based APCA techniques?

　RQ1.2: How does APPT perform compared with existing state-of-the-art traditional and learning-based APCA techniques?

RQ2 (Impact analysis): To what extent do the different choices affect the overall effectiveness of APPT?

　RQ2.1: To what extent do the training choices affect the overall effectiveness of APPT?

　RQ2.2: To what extent do the vector concatenation choices affect the overall effectiveness of APPT?

　RQ2.3: To what extent do the pre-trained model choices affect the overall effectiveness of APPT?

RQ3 (Cross-project effectiveness): How does APPT perform on the new projects in the cross-project prediction scenario?

RQ1 aims to investigate the effectiveness of APPT, which is further refined into two sub-RQs. In detail, RQ1.1 explores to what extent APPT outperforms existing representation learning techniques, including three classifiers multiplied (decision tree, logistic regression, and naive Bayes) by five representation methods (BERT, code2vec, code2seq, Doc2Vec, and CC2Vec) from Tian et al. [16], and the most recent technique CACHE from Lin et al. [28]. RQ2.2 explores the effectiveness of APPT by comparing it with both dynamic and static techniques. The latest learning-based APCA technique, ODS, is also evaluated in our study.

RQ2 focuses on impact analysis of APPT, which is further refined into three sub-RQs. In detail, RQ2.1 explores how the training choices affect the effectiveness of APPT. RQ2.2 explores how the five vector concatenation methods affect the effectiveness of APPT. RQ2.3 replaces BERT with advanced CodeBERT and GraphCodeBERT to investigate the impact of the pre-trained models on the effectiveness of APPT. RQ3 explores the effectiveness of APPT when predicting patches from unseen projects.

4.2 Dataset

In this study, we adopt two patch datasets based on the recent studies [16], [17], [28], a small one containing 1,183 Defects4J labeled patches and a large one containing 50,794 real-world labeled patches.

TABLE 1: Statistics of patches in the small dataset

#Tools	#Overfitting	#Correct	#Total
Nopol	29	1	30
jGenProg	16	2	18
GenProg	24	1	25
SketchFix	7	5	12
defects4j-dev	0	354	354
DynaMoth	21	1	22
CapGen	41	9	50
TBar	33	7	40
SOFix	1	10	11
FixMiner	19	6	25
ACS	5	16	21
Kali	36	2	38
kPAR	32	2	34
Jaid	40	32	72
AVATAR	37	17	54
SequenceR	45	10	55
Arja	49	8	57
RSRepair	31	2	33
ICSE18	99	28	127
jKali	18	4	22
SimFix	42	16	58
jMutRepair	14	2	16
Cardumen	9	0	9
Total	648	535	1183

On the small dataset, we mainly focus on the released patches from Defects4J [42], which is the most widely-adopted benchmark in APR research [9]. We select the datasets released by two recent large-scale studies, i.e., Wang et al. [17] and Tian et al. [16]. Specifically, the first benchmark [17] includes the labeled patches provided by Liu et al. [9], Xiong et al. [21] and Defects4J developers [42]. The second benchmark [16] includes the labeled patches from Liu et al. [9] and also considers the patches generated by some well-known APR tools that are not included in Liu et al. [9] to better explore the overfitting problem, i.e., JAID [43], SketchFix [44], CapGen [45], SOFix [46] and SequenceR [47]. To avoid the data leakage issue in the two benchmarks, a filtering process is also conducted to discard duplicate patches. In particular, given a patch whose all the blank spaces are removed, the left text information is compared with that from the other patches. If two patches are identical concerning their text information, they are considered duplicates, resulting in 1,183 patches in our small dataset. The patches are generated by 22 distinct APR tools, which can be divided into four categories, i.e., heuristic-based,

TABLE 2: Datasets used in our experiment

Datasets	Subjects	# Correct	# Overfitting	Total
Small	Tian et al. [16]	468	532	1,000
	Wang et al. [17]	248	654	902
	Our Study	535	648	1,183
Large	ManySStuBs4J [48]	51,433	0	51,433
	RepairThemAll [8]	900	63,393	64,293
	Our Study	25,589	24,105	49,694

constraint-based, template-based, and learning-based techniques. The detailed information on these covered APR tools is presented in Table 1, where the first column lists the four repair technique categories and the second column list the corresponding repair techniques.

On the large dataset, we further consider a variety of patches generated from other benchmarks, to evaluate the generality of APPT. Recently, existing studies demonstrate that APR techniques may overfit Defects4J in terms of repairability [8], [14]. Thus, some other benchmarks have been applied to evaluate the performance of APR techniques, such as Bugs.jar [49], IntroclassJava [50], BEARS [51] and QuixBugs [52], providing substantial patches on the large dataset. In this work, we consider a large patch dataset released by a recent study [28] to investigate the generality of APPT. The large patch dataset includes the labeled patches provided from RepairThemAll framework [8] and ManySStuBs4J [48]. In particular, RepairThemAll framework [8] contains 64,293 patches using 11 Java test-suite-based repair tools and 2,141 bugs from five diverse benchmarks. However, Tian et al. [16] manually inspect the correctness of the patches from RepairThemAll[2] and find 98.6% of patches (63,393/64,293) are actually overfitting ones, resulting in an imbalanced dataset. Recent studies have revealed that a well-balanced dataset is essential when investigating deep learning-based prediction techniques [16], [22]. To compensate the lack of correct patches, the large patch dataset then includes ManySStuBs4J [48], which provides simple bug-fix changes mined from 1,000 popular open-source Java projects. The bug-fix changes are correct fix attempts of real-world bugs and thus are considered correct patches in our experiment. Finally, a large balanced patch dataset is built from the RepairThemAll framework and ManySStuBs4J by discarding duplicate patches and filtering the ones from small student-written programming assignments (e.g., IntroClassJava). The dataset involves all available patches generated on RepairThemAll framework and ManySStuBs4J, resulting in 49,694 patches after deduplication.

Statistics on the two datasets are shown in Table 2. Table 2 has two main rows representing the two datasets, each of which has three sub-rows. The first and second sub-rows list the two sources in the corresponding dataset. The third column lists the filtered patches used in our experiment from the two sources. We also present the number of correct, overfitting and total patches in the last three columns.

---

2. The RepairThemAll Framework. https://github.com/program-repair/RepairThemAll, accessed March 2023

4.3 Baselines

Various APCA techniques have been proposed in the literature to validate patch correctness. Following existing studies [21], [28], we attempt to select state-of-the-art techniques designed for Java language as Java is the most targeted language in APR community [9] and the existing patches of real-world bugs are usually available in Java language [16]. We first consider the recent empirical study by Wang et al. [17] to identify existing APCA techniques. We then select recent advanced studies [16], [28] that are not included in Wang et al. [17].

In general, following existing work [17], [28], [57], [58], the existing APCA techniques can be categorized into static, dynamic and learning-based APCA techniques according to whether test execution is needed or deep learning techniques are adopted (mentioned in Section 2). Meanwhile, according to whether the ground-truth patch is required, they can be further categorized into two categories (i.e., with or without oracle). Particularly, similar to our proposed method APPT, CACHE and embedding learning techniques adopt representation models to embed changed code and a deep learning classier to predict patch correctness. Such techniques can be further considered as representation learning APCA techniques.

The details of the selected APCA techniques are illustrated in Table 3. The first column lists three APCA categories. The second and third columns list whether the oracle information is equipped. We also list the representation learning techniques (e.g., APPT) in the light gray box. We summarize the selected techniques as follows.

4.3.1 Dynamic-based APCA Techniques

Dynamic-based techniques are designed to distinguish correct patches from overfitting patches based on the outcome or the execution traces of the original or generated test cases.

Simple Test Generation. The overfitting issue is prevalent in the repair process due to the weak adequacy of existing test cases. Thus, researchers use test case generation tools to generate extra test cases based on the fixed program and check whether or not the generated patches that pass the original test cases can pass the extra test cases [27], [59]. In this work, we adopt Evosuite [19] and Randoop [20] as the test case generation tools, as they have been widely investigated in previous studies.

DiffTGen. Xin et al. [53] identify overfitting patches by executing test cases generated by an external test generator (i.e., Evosuite). Different from *simple test generation* generating test cases randomly, DiffTGen generates test cases to uncover the syntactic differences between the patched and buggy program. A plausible patch is regarded as overfitting if the output of the patched program is not the same as that of the correct program. DiffTGen needs a human-written patch as a reference and requires providing human-amenable testing information for the developers to provide oracles the generated test cases.

Daikon. Daikon is a dynamic-based technique based on the program invariant with oracle information. Yang et al. [54] adopt the program invariant to explore the differences between an overfitting and a correct patch. A patch is considered correct if its inferred invariant is identical to that

TABLE 3: Compared APCA techniques in our experiment

	with Oracle Required	without Oracle Required
Dynamic-based	Evosuite [19], Randoop [20], DiffTGen [53], Daikon [54]	PATCH-SIM [21], E-PATCH-SIM [21], R-Opad [54], E-Opad [54]
Static-based	⊗	ssFix [55], CapGen [45], Anti-patterns [18], S3 [56]
Learning-based	⊗	ODS [22], Random Forest [16], Embedding learning [16], CACHE [28], Our proposed APPT

▮ denotes the representation learning techniques.

of the ground-truth. If there exists a different comparison, the patch is considered overfitting.

PATCH-SIM. Xiong et al. [21] consider the execution traces of the passing tests on the buggy and patched programs are likely to be similar, while the execution traces of failing tests on the buggy and patched programs are likely different. Based on the concept, they approximate the correctness of a patch based on the execution trace without the oracle information. PATCH-SIM adopts Randoop to generate additional test cases to collect dynamic execution information. In this work, we also replace Randoop with Evosuite to comprehensively explore the impact of test generation techniques (denoted as E-PATCH-SIM).

Opad. Yang et al. [60] adopt fuzzing testing to generate new test cases and employ two test oracles (crash and memory-safety) to enhance the validity checking of patches. The original implementation of Opad is not designed for Java language and uses American Fuzz Lop (AFL) as the fuzzing technique. In this work, following recent studies [17], [28], we replace AFL with Randoop and Evosuite to generate new test cases on the Java programs and denote them as R-Opad and E-Opad, respectively.

4.3.2 Static-based APCA Techniques

Static-based techniques usually adopt static analysis tools to extract some designed static features and then check patch correctness based on such features.

ssFix. ssFix [55] is a static-based technique that utilizes token-based syntax representation to generate patches with a higher probability of correctness. ssFix first performs a syntactic code search to find code snippets from a codebase that is syntax-related to the context of a bug to generate correct patches, and then prioritizes the patches based on the modification types and the modification sizes.

CapGen. Wen et al. [45] propose three aspects of context information (i.e., genealogy contexts, variable contexts and dependency contexts) embedded in an AST node and its surrounding codes to prioritize correct patches over overfitting ones. In this work, following recent studies [17], [28], we extract the three context information as static features to investigate patch correctness assessment.

Anti-patterns. Tan et al. [18] define a set of rules that essentially capture disallowed modifications to the buggy program, and a patch is overfitting if it falls into the rules. A recent study [17] has shown that the manually-defined anti-patterns may have false positives for correct patches, resulting in destructive effects in patch correctness prediction.

S3. Le et al. [56] assume that a correct patch is often syntactically and semantically close to a buggy code snippet.

Thus, they adopt six syntactic features (i.e., AST differencing, cosine similarity and locality of variables and constants) and semantic features (i.e., model counting, output coverage and anti-patterns) to measure the distance between a candidate patch and the buggy code snippet.

4.3.3 Learning-based APCA Techniques

Learning-based techniques mainly investigate static features and machine learning techniques to build predictive models for patch correctness prediction.

ODS. Ye et al. [22] first extract 202 code features at the abstract syntax tree level and then use supervised learning to learn a probabilistic model automatically. The results show that ODS can achieve better prediction performance than the dynamic-based technique PATCH-SIM with a faster speed.

CACHE. Lin et al. [28] propose a context-aware APCA technique CACHE by taking both the changed code snippet and the correlated unchanged code snippet into consideration. CACHE first parses the patched code snippet into AST representation and then utilizes the AST path technique to capture the structure information.

Random Forest. Wang et al. [17] investigate the effectiveness of adopting deep learning models to predict patch correctness based on eight static features (two from ssFix, three from S3, and three from CapGen). To integrate the static features, six widely-used classification models (including Random Forest, Decision Table, J48, Naive Bayes, Logistic Regression, and SMO) are adopted. The results demonstrate that Random Forest can achieve both superior precision and recall performance. In this work, following existing work [28], we also adopt Random Forest to predict the patch correctness based on the integrated static features.

Embedding Learning. Tian et al. [16] propose to leverage representation learning techniques to produce embedding for buggy and patched code snippets and then adopt supervised learning classifies to predict patch correctness. In particular, nine representation learning APCA techniques are evaluated, involving three embedding techniques (i.e., CC2vec, BERT and Doc2Vec) and three classifiers (logistic regression, decision tree and naive bayes). APPT differs from Tian et al. [16] in that BERT is adopted as one of the embedding techniques to embedding code without any training, while we attempt to take advantage of the generic knowledge of the pre-train model and the classifier by further fine-tuning APPT to support patch correctness assessment. Considering the fact that off-the-shelf pre-trained models (e.g., BERT) are pre-trained with data corpora in other fields (e.g., NLP) and thus may not be suitable to embed patched code snippets, fine-tuning pre-trained model

in our APPT architecture can obtain optimal embedding vectors for reasoning about patch correctness.

4.4 Model Selection

To the best of our knowledge, APPT is the first automated patch correctness prediction technique by fine-tuning the existing pre-trained model in the APCA domain. In this paper, we adopt BERT as the encoder stack due to its powerful performance in previous work [29].

Specifically, BERT is pre-trained on large amounts of text data with two self-supervised goals, i.e., masked language modeling (MLM) and next sentence prediction (NSP). MLM aims to let the model predict the masked words by masking 15% of words in each sentence randomly. NSP aims to further improve the model's ability to understand the relationship between two sentences by letting the model predict whether the given sentence pair is continuous. The model then can be fine-tuned to adapt to some specific downstream tasks and has achieved remarkable state-of-the-art results on a variety of natural language processing tasks, such as question answering and language inference.

There exist two model architectures at different sizes, i.e., $BERT_{base}$ and $BERT_{large}$ [29]. The former has 12 layers and 12 attention heads, and the embedding size is 768, while the latter has a double layer number and 16 attention heads, and the embedding size is changed to 1024. In this paper, we do not modify the vocabulary size and use the pre-trained $BERT_{base}$ as the fine-tuning starting point instead of starting from scratch.

In this paper, APPT is conceptually and practically generalizable to various pre-trained models. We also select CodeBERT and GraphCodeBERT as the encoder stack to evaluate the scalability of APPT. CodeBERT and GraphCodeBERT share the same model architecture as BERT, while utilizing paired natural language and programming language to pre-train the model to support code-related tasks (mentioned in Section 8.2.2).

4.5 Evaluation Metrics

We evaluate the prediction performance of various APCA approaches by accuracy, precision, recall, F1-score and AUC metrics, which have been widely adopted in patch correctness assessment research and other classification tasks [16], [28]. Given the number of true positives (TPs, a TP refers to an overfitting patch that is identified as overfitting), false positives (FPs, a FP refers to a correct patch that is identified as overfitting), false negatives (FNs, a FN refers to an overfitting patch is identified as correct) and true negatives (TNs, a TN refers to a correct patch that is identified as correct), the metrics are defined as follows:

• *Accuracy:* the proportion of correctly reported (whether the patch is correct or not) patches. Accuracy measures the probability that the prediction of APCA techniques is correct.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \quad (12)$$

• *Precision:* the proportion of real overfitting patches over the reported overfitting patches. Precision measures how much we can trust the APCA techniques when it predicts a patch as overfitting.

$$Precision = \frac{TP}{TP + FP} \quad (13)$$

• *Recall:* the proportion of reported overfitting patches over all the real overfitting patches. Recall measures the ability of the APCA techniques to find all the overfitting patches in the dataset.

$$Recall = \frac{TP}{TP + FN} \quad (14)$$

• *F1-score:* twice the multiplication of precision and recall divided by the sum of them. F1-score measures the trade-off between precision and recall by taking their harmonic mean.

$$F1\text{-}score = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (15)$$

• *AUC:* the entire two-dimensional area underneath the entire receiver operating characteristic curve. AUC measures the probability that the classifier will rank a randomly chosen overfitting patch higher than that of a randomly chosen correct patch. The higher the AUC, the better the APCA techniques is at predicting real overfitting patches as overfitting and real correct patches as correct.

$$AUC = \frac{\sum_{i \in O} \text{rank}_i - M(M+1)/2}{M \times N} \quad (16)$$

where M and N denote the number of overfitting and correct patches, respectively. O denotes the overfitting patch set and $rank_i$ denotes the rank of the $i-$th overfitting patch in the descending list of output probability produced by each model.

4.6 Implementation Details

All of our approaches are built based on PyTorch framework[3]. We use the Hugging Face[4] implementation version of BERT in our work. Considering previous work recommendation [31], [61], we utilize "bert-base-uncased" (refer to $BERT_{base}$) as the initial point, as the base version is quite lightweight to employ in practice with comparable effectiveness compared against the large version. There exist 12 layers of transformer blocks and 12 self-attention heads in the "bert-base-uncased" model. The optimizer is Adam [41] with $5e - 5$ learning rate. The batch size is 16 and dropout rate is 0.5. We train for most 50 epochs and the max length of the input is set to 512 due to model limitation.

All the training and evaluation of our methods are conducted on one Ubuntu 18.04.3 server with two Tesla V100-SXM2 GPUs.

5 RESULTS AND ANALYSIS

5.1 RQ1: Effectiveness of APPT

5.1.1 Comparing with Representation Learning-based Techniques

Experimental Design. As discussed in Section 4.3, APPT, CACHE and embedding learning techniques (i.e., tech-

---

3. PyTorch. https://pytorch.org/, accessed March 2023
4. Hugging Face. https://huggingface.co/, accessed March 2023

TABLE 4: Effectiveness of APPT compared with representation learning-based APCA techniques on the small dataset

Classifier	Embedding	Accuracy	Precision	Recall	F1-score	AUC
Decision Tree	BERT	63.5%	65.3%	70.9%	67.9%	63.7%
	CC2vec	66.1%	69.4%	68.0%	68.7%	66.5%
	code2vec	65.1%	68.1%	68.3%	68.1%	64.4%
	code2seq	60.1%	63.5%	64.0%	63.7%	60.0%
	Doc2Vec	61.2%	64.5%	65.3%	64.8%	60.8%
Logistic Regression	BERT	64.8%	66.5%	72.4%	69.2%	68.7%
	CC2vec	64.9%	62.4%	90.1%	73.7%	68.6%
	code2vec	66.8%	68.6%	72.9%	70.6%	70.2%
	code2seq	60.7%	63.3%	67.6%	65.3%	63.1%
	Doc2Vec	63.7%	65.7%	70.8%	68.0%	68.9%
Naïve Bayes	BERT	61.6%	64.8%	65.7%	65.0%	64.7%
	CC2vec	60.0%	58.3%	**94.6%**	72.2%	58.1%
	code2vec	57.7%	58.1%	81.5%	67.8%	55.6%
	code2seq	57.0%	59.0%	70.5%	64.2%	60.6%
	Doc2Vec	64.1%	65.8%	72.4%	68.7%	67.0%
CACHE		75.4%	79.5%	76.5%	78.0%	80.3%
APPT		**79.7%**	**80.8%**	83.2%	**81.8%**	**82.5%**

niques within the light gray box in Table 3) can be categorized as representation learning APCA techniques. In this section, we aim to explore the performance of APPT when compared with these representation learning techniques. In particular, embedding learning techniques [16] mainly adopt embedding models (i.e., BERT, Doc2Vec, and CC2Vec) to embed buggy and patched code fragments, and then train classification models (i.e., Decision Tree, Logistic Regression, and Naive Bayes) to predict patch correctness. Following previous study [28], we also consider two additional embedding models (i.e., code2vec and code2seq) in the experiment. Meanwhile, CACHE can also be considered as a representation learning technique, which incorporates the context information in embedding code changes, and trains a deep learning classifier to predict the patch correctness.

In total, one representation learning technique with 15 settings from Tian et al. [16] (involving five embedding techniques multiplied by three classification models), and one context-aware representation learning technique CACHE are considered in our experiment. Following the previous study [16], we perform standard practice 5-fold cross-validation on both the small and large datasets for comparison.

Results. Comparison results against the existing representation learning techniques are presented in Table 4 to Table 5 for both the small and large datasets. The first column lists the three classifiers and the second column lists the five off-the-shelf embedding models. The remaining columns list the detailed values of accuracy, precision, recall, F1-score and AUC metrics, respectively. We present the most recent representation learning work CACHE and our APPT in the bottom part of Table 4 and Table 5. It can be observed that APPT achieves the best performance under each experimental setting.

On the small dataset, APPT is around 4.3%, 1.3%, 6.7%, 3.8% and 2.2% higher than the state-of-the-art technique CACHE in terms of all metrics (i.e., 79.7% vs. 75.4% for accuracy, 80.8% vs. 79.5% for precision, 83.2% vs. 76.5% for recall, 81.8% vs. 78.0% for F1-score, and 82.5% vs. 80.3% for AUC). Compared with all representation learning techniques, APPT achieves the best performance in terms of

accuracy, precision, F1-score and AUC metrics. In particular, the values of APPT on the accuracy and precision metrics are 79.7% and 80.8%, respectively, while the optimal values of all other techniques are 75.4% and 79.5%. This suggests that APPT can generally achieve the most accurate predictions, and the patches identified as overfitting by APPT are of high confidence to be overfitting. Regarding recall, the values of CC2vec and code2vec can sometimes exceed those of APPT since they tend to classify most patches as overfitting (e.g., CC2vec with Naive Bayes classifies 1,051 out of 1,183 patches as overfitting and thus achieves a high recall of 94.6%). However, these techniques achieve relatively low precision (e.g., CC2vec with Naive Bayes classifier has only 72.2% for recall). In contrast, APPT can achieve a high recall exceeding 83% while maintaining a high precision of 80.8%.

On the large dataset, we can find APPT achieves over 99% for the five metrics, outperforming all existing approaches. For example, APPT reaches 99.9% in terms of AUC, which is 1.0% higher than the second highest value obtained from the most recent technique CACHE (i.e., 98.9%). This suggests that APPT is more capable of distinguishing correct and overfitting patches than CACHE. Besides, the improvement against CACHE for accuracy, precision, recall and F1-score metrics achieves 0.5%, 0.3%, 0.9% and 0.5%, respectively. We also find that the performance achieved on the large dataset is commonly higher than that achieved on the small dataset. For example, the average value among the five metrics increases from 81.06% to 99.26%, resulting in a 22.5% improvement rate. Based on our analysis on the two datasets, the possible reason for this improvement is that bugs on the large dataset are usually simple. We observe that all ManySStuBs4J patches on the large dataset are single-line operations, while patches on the small dataset usually cross multiple lines (e.g., more than 40% of Defects4J developer patches are multiple line patches [28]). It is easy for neural networks to learn the correctness distribution of such simple code changes. Meanwhile, the difference in patch scale between the two datasets may be the second reason. We find there exist 49,694 patches on the large dataset, which is 42 times larger than that of the

TABLE 5: Effectiveness of APPT compared with representation learning techniques on the large dataset

Classifier	Embedding	Accuracy	Precision	Recall	F1-score	AUC
Decision Tree	BERT	95.7%	93.9%	97.4%	95.6%	95.9%
	CC2vec	95.6%	95.4%	95.7%	95.5%	95.7%
	code2vec	95.0%	93.2%	96.6%	94.9%	95.4%
	code2seq	92.2%	91.0%	93.2%	92.3%	92.4%
	Doc2Vec	85.1%	84.2%	85.3%	84.7%	85.3%
Logistic Regression	BERT	82.4%	83.6%	79.4%	81.4%	91.0%
	CC2vec	91.2%	96.1%	85.4%	90.4%	95.0%
	code2vec	89.6%	88.6%	90.2%	89.4%	95.0%
	code2seq	91.5%	90.5%	92.2%	91.4%	96.0%
	Doc2Vec	90.4%	91.9%	88.0%	89.9%	96.1%
Naïve Bayes	BERT	68.2%	80.3%	45.7%	58.2%	74.6%
	CC2vec	78.4%	94.8%	58.6%	72.5%	92.4%
	code2vec	61.4%	68.7%	37.4%	48.4%	69.3%
	code2seq	70.3%	76.8%	55.5%	64.5%	78.9%
	Doc2Vec	81.2%	86.4%	75.5%	78.9%	88.9%
CACHE		98.6%	98.9%	98.2%	98.6%	98.9%
APPT		**99.1%**	**99.1%**	**99.1%**	**99.1%**	**99.9%**

small dataset. The amount of training data is often the single most dominant factor that determines the performance of the neural networks [62]. More available patches benefit the neural networks to learn diverse code changes better.

Case study of unique identified overfitting and correct patch. Fig. 3 presents an example of an overfitting patch generated for bug Math-53, which is detected as overfitting by APPT but not CACHE and Tian et al. [16]. In this bug, the method *add()* (line 1 in P_3) is used to return a Complex whose value is *this + rhs* (*rhs* is the parameter value to be added to this Complex). Ideally, if either *this* or *rhs* has a NaN value in either part, NaN is returned; otherwise Infinite and NaN values are returned in the parts of the result according to the rules for Double arithmetic. In the buggy version, *add()* directly creates a complex number given the real and imaginary parts. Consequently, the buggy version fails to check whether or not the parameter has a NaN value and throws AssertionFailedError. As we find in P_1, Jaid fixes the bug by adding the condition *if ((isNaN() || rhs.isNaN()) == true)* (line 4 in P_1), which is equivalent the condition written by developers *if (isNaN || rhs.isNaN)* (line 4 in P_3). However, CACHE fails to detect the semantic equivalence between the developer patch and the correct patch. In contrast, APPT, which relies on a pre-trained model and fine-tuning process, still can correctly identify the correct patch. Similarly, another example can be seen in P_2, in which an overfitting patch generated by HDRepair cannot be detected by CACHE but by APPT as APPT successfully captures the different behaviors between the expression *rhs.getArgument()* (line 5 in P_2) and *rhs.getArgument()* (line 5 in P_3).

```
A Correct Patch P1 Generated by Jaid
1       public Complex add(Complex rhs)
2           throws NullArgumentException {
3           MathUtils.checkNotNull(rhs);
4   +       if((isNaN() || rhs.isNaN()) == true)
    {
5   +           return NaN;
6   +       }
7           return createComplex(real +
    rhs.getReal(),
8               imaginary + rhs.getImaginary());
9       }
```
```
A Plausible Patch P2 Generated by HDRepair
1       public Complex add(Complex rhs)
2           throws NullArgumentException {
3           MathUtils.checkNotNull(rhs);
4   -       return createComplex(real +
    rhs.getReal(),
5   +       return createComplex(real +
    rhs.getArgument(),
6               imaginary + rhs.getImaginary());
7       }
```
```
A Correct Patch P3 Generated by Developers
1       public Complex add(Complex rhs)
2           throws NullArgumentException {
3           MathUtils.checkNotNull(rhs);
4   +       if (isNaN || rhs.isNaN) {
5   +           return NaN;
6   +       }
7           return createComplex(real +
    rhs.getReal(),
8               imaginary + rhs.getImaginary());
9       }
```

Fig. 3: APR-generated and developer patches for Math-53

5.1.2 Comparing with APCA Techniques

Experimental Design. In this section, we aim to further compare the proposed method APPT with the existing APCA techniques. We select the remaining techniques mentioned in Section 4.3 (except representation learning techniques discussed in RQ1). In total, 14 APCA techniques are considered in the experiment, involving four static techniques (Anti-patterns, ssFix, CapGen and S3), eight dynamic techniques (Evosuite, Randoop, DiffTGen, Daikon, R-Opad, E-

Answer to RQ1.1: Overall, our analysis on representation learning techniques reveals that (1) APPT can outperform a state-of-the-art representation learning technique CACHE under all metrics and datasets. (2) on the small dataset, APPT achieves 79.7% for accuracy and 82.5% for AUC, which surpass CACHE by 4.3% and 2.2%. (3) on the large dataset, APPT exceeds 99% on all metrics, yet none of existing representation learning techniques achieves that.

TABLE 6: Effectiveness of APPT compared with the traditional and learning-based APCA technique

Category		APCA	Accuracy	Precision	Recall.	F1-score
Dynamic-based	w-oracle	Evosuite	65.9%	99.1%	53.5%	69.5%
		Randoop	51.3%	97.4%	33.8%	50.2%
		DiffTGen	49.6%	97.4%	30.6%	46.6%
		Daikon	76.1%	89.9%	73.7%	81.0%
	wo-oracle	R-Opad	34.9%	**100.0%**	10.2%	18.5%
		E-Opad	37.7%	**100.0%**	14.7%	25.6%
		PATCH-SIM	49.5%	83.0%	38.9%	53.0%
		E-PATCH-SIM	41.7%	82.1%	25.8%	39.3%
Static-based		Anti-patterns	47.6%	85.5%	33.5%	48.1%
		S3	69.7%	79.3%	78.9%	79.0%
		ssFix	69.2%	78.9%	78.8%	78.8%
		CapGen	68.0%	78.3%	77.4%	77.8%
Learning-based		Random Forest	72.5%	87.0%	89.1%	88.0%
		ODS	88.9%	90.4%	94.8%	92.5%
APPT			**90.4%**	91.5%	**96.0%**	**93.6%**

Opad, PATCH-SIM and E-PATCH-SIM) and two learning techniques (Random Forest and ODS).

As it is time-consuming to run all the techniques (especially for dynamic and learning ones), following the existing work [28], we reuse the released results from the recent work [17], [22], [28]. We collect the detailed results of all selected APCA techniques from Lin et al. [28], which are concluded based on 902 patches (i.e., Wang et al. [17] in Table 2) and a 10-fold cross-validation. To fairly compare with all the state-of-the-art techniques, we perform our experiment in the same experimental setting.

Results. The experiment results are listed in Table 6. The first two columns list the selected techniques and their corresponding categories. The remaining columns list the detailed values of accuracy, precision, recall and F1-score metrics.

Compared with traditional dynamic-based and static-based APCA techniques, we can find that APPT reaches 90.4%, 96.0% and 93.6% in terms of accuracy, recall and F1-score, respectively. Specifically, APPT achieves the best overall performance with the three metrics, and none of the previous techniques exceeds 90%. As for precision, more than 91% of patches reported by APPT are indeed overfitting patches, which is better than all static-based techniques and three dynamic-based techniques (i.e., Daikon, PATCH-SIM, and E-PATCH-SIM). Although some dynamic ones have higher precision values, it is time-consuming to generate additional test cases and collect run-time information. More importantly, the recall of these techniques is usually low (e.g., 10.3% for R-Opad), or the ground-truth oracle is needed (e.g., Evosuite and Randoop techniques), limiting the application of such techniques in practice.

Compared with learning-based techniques, we find that APPT still performs better than a state-of-the-art technique ODS with respect to all four metrics (90.4% vs. 88.9% for accuracy, 91.5% vs. 90.4% for precision, 96.0% vs. 94.8% for recall, 93.6% vs. 92.5% for F1-score, respectively). Overall, the improvement against Random Forest and ODS reaches 4.5%~17.9% and 1.1%~1.5%. Considering that it is expensive for ODS to extract hundreds of manually-designed code features at AST level, our approach simply adopting the pre-trained model to encode a sequence of tokens is

even more promising. We also highlight this direction of integrating code-aware features (e.g., code edits and AST representation) with pre-trained models for patch correctness assessment.

> **Answer to RQ1.2:** Overall, our comparison results reveal that, (1) APPT can achieve remarkable performance compared to exiting static-based techniques with a high recall reaching 96.0%; (2) APPT can achieve higher precision than a state-of-the-art dynamic-based technique PATCH-SIM by 8.5%; (3) compared with existing learning-based techniques, APPT can achieve the best performance among all metrics.

5.2 RQ2: The Impact Analysis

In this section, we further explore how different experiment choices affect the prediction performance of APPT.

5.2.1 The impact of Training Choice

Experimental Design. APPT employs a pre-trained language model as the encoder stack, which is connected with an LSTM stack for classification training. The quality of vector representation heavily relies on the language models of code being used. In this process, the pre-trained model is further fine-tuned to obtain a suboptimal vector representation of code for patch correctness assessment. Thus, we formulate this subRQ to investigate the impact of the pre-training, fine-tuning, and LSTM components.

Results. Table 7 presents the ablation study results under different training choices. The first column lists the two datasets. The second column lists the three training choices, i.e., without pre-training, fine-tuning, and LSTM components. The remaining columns list the detailed values of accuracy, precision, recall and F1-score and AUC metrics.

Generally speaking, all training components make contributions to the performance of APPT in terms of these metrics. For example, if the LSTM stack is not included on the small dataset, the accuracy and recall of APPT will be decreased by 1.64% and 7.02%. This finding demonstrates the rationale of our motivation that the LSTM stack is suitable to extract more hidden code change features for patch correctness assessment.

TABLE 7: Effectiveness of APPT with different training choices

Dataset	Component	Accuracy	Precision	Recall	F1-score	AUC
Small	$APPT_{pre-training-}$	72.46%(↑7.26%)	69.28%(↑11.56%)	89.15%(↑-5.98%)	77.97%(↑3.8%)	82.24%(↑0.31%)
	$APPT_{fine-tuning-}$	69.63%(↑10.09%)	67.67%(↑13.16%)	81.35%(↑1.82%)	71.54%(↑10.22%)	71.77%(↑10.78%)
	$APPT_{LSTM-}$	76.84%(↑2.88%)	79.2%(↑1.64%)	76.15%(↑7.02%)	77.65%(↑4.11%)	80.65%(↑1.90%)
	APPT	79.72%	80.84%	83.17%	81.76%	82.55%
Large	$APPT_{pre-training-}$	98.%(↑1.14%)	98.62%(↑0.47%)	97.24%(↑1.89%)	97.91%(↑1.19%)	99.46%(↑0.40%)
	$APPT_{fine-tuning-}$	98.29%(↑0.85%)	98.72%(↑0.37%)	97.74%(↑1.39%)	98.23%(↑.88%)	99.74%(↑0.12%)
	$APPT_{LSTM-}$	86.66%(↑12.48%)	85.94%(↑13.15%)	86.7%(↑12.42%)	86.31%(↑12.8%)	93.35%(↑6.51%)
	APPT	99.13%	99.09%	99.13%	99.11%	99.86%

We find that the fine-tuning component contributes the most to the overall performance of APPT without which the Precision will degrade the most for both datasets. For instance, if we do not fine-tune the pre-trained model, the precision of APPT will be decreased by 13.16% on the small dataset. This finding demonstrates the rationale of our motivation that fine-tuning the pre-trained model can help better convert the code snippets into the embedding, which is quite different from Tian et al. [16]. We also note if we do not employ the pre-training knowledge (i.e., training the pre-trained model from scratch), the performances will drop at notable degrees (e.g., 7.26% and 11.56% for accuracy and precision on the small dataset). This finding highlights the substantial benefits of the pre-training process on the larger corpus to assess the correctness of a patch.

When comparing the improvements in the two datasets, we find that the improvement on the large datasets is not as significant as on the small dataset. We think this phenomenon is quite reasonable due to the differences between the two datasets. The bugs in the large dataset are usually simpler than those in the small dataset and can be fixed with single-line operations (discussed in Section 5.1). As a result, APPT has already achieved very high results on the large dataset (e.g., exceeding 99% in all five metrics), which leads to the difficulty in obtaining even greater improvements. Compared to the large dataset, we believe that the small dataset is able better to reflect the true capability of our approach and baselines, as the small dataset is constructed from Defects4J, which contains a variety type of real-world bugs and is the most widely used benchmark in the field of program repair and patch correctness assessment. From Table 7, we observe a significant improvement in the dataset, which is more valuable to demonstrate the significance of our three training components.

5.2.2 The Impact of The Vector Concatenation Choice

Experimental Design. In the vector integration process, APPT directly concatenates the buggy method vector and patched method vector. We formulate this subRQ to investigate the impact of different integration methods, such as concatenate additional, subtraction, and product operation (detailed in Section 3.3). We also consider the vector integration methods used in Tian et al. [16] and CACHE [28].

Results. Table 8 presents the prediction results under different concatenation choices. The first column lists the two datasets. The second column lists the six concatenation choices, i.e., addition, subtraction, product, and the one used in CACHE and Tian et al. and APPT. The remaining columns list the detailed values of accuracy, precision, recall, F1-score and AUC metrics.

On the small dataset, although conceptually simple, $APPT_{concat}$ can obtain 79.72%, 80.84%, 83.17%, 81.76%, and 82.55% for accuracy, precision, recall, F1-score and AUC metrics, four of which are highest among all investigated concatenation methods. $APPT_{product}$ has the highest recall score (96.32%), while it performs worse than $APPT_{concat}$ by 16.45%, 18.47%, 6.95% and 16.09% for the other four metrics. $APPT_{addition}$ and $APPT_{subtraction}$ perform the addition and subtraction operation for buggy and patched vectors, and have similar performance for all metrics. Meanwhile, mixed methods (i.e., $APPT_{CACHE}$ and $APPT_{Tian\ et\ al.}$) that apply different comparison functions to represent the changed embedding vector can achieve comparable results with $APPT_{concat}$, which is also consistent with the existing study results [16], [63]. On the large dataset, $APPT_{concat}$ achieves the best performance in accuracy, F1-score and AUC metrics, while $APPT_{CACHE}$ and $APPT_{Tian\ et\ al.}$) perform best in precision and recall respectively. The difference in performance is similar as the methods have relatively high metric values. For example, all metric values are higher than 99% for $APPT_{concat}$ and $APPT_{CACHE}$.

5.2.3 The Impact of Pre-trained Model Choice

Experimental Design. Recently, following the BERT model architecture, researchers use some code-related pre-trained tasks to capture the semantic connection between natural language and programming language, so as to further adapt these pre-training models for programming language. Thus, we formulate this subRQ to investigate the impact of different pre-trained models by replacing the BERT with two advanced models pre-trained with the programming language, i.e., CodeBERT [34] and GraphCodeBERT [35].

Results. Table 9 demonstrates the predicted performance of three pre-trained models. The first column lists the two datasets. The second column lists the three models, i.e., BERT, CodeBERT, and GraphCodeBERT. The remaining columns list the detailed values of accuracy, precision, recall, F1-score, and AUC metrics.

Generally speaking, all of the adopted models achieve a higher performance than the state-of-the-art technique CACHE on all metrics. For example, on the small dataset, BERT, CodeBERT, and GraphCodeBERT reach 81.76%, 83.35%, and 83.47% with respect to the F1-score, which is 3.76%, 5.35%, and 5.47% higher than CACHE, respectively. A similar improvement can also be observed on the large

TABLE 8: Effectiveness of APPT with different concatenation choices

Dataset	Concatenation	Accuracy	Precision	Recall	F1-score	AUC
Small	$APPT_{addition}$	69.83%	70.24%	80.12%	73.83%	75.44%
	$APPT_{subtraction}$	71.38%	72.42%	77.27%	74.72%	75.59%
	$APPT_{product}$	63.27%	62.37%	**96.32%**	74.81%	66.46%
	$APPT_{CACHE}$	78.36%	79.33%	81.95%	80.60%	81.57%
	$APPT_{Tian\ et\ al.}$	78.19%	78.12%	84.09%	80.86%	81.67%
	$APPT_{concat}$	**79.72%**	**80.84%**	83.17%	**81.76%**	**82.55%**
Large	$APPT_{addition}$	98.96%	98.80%	99.07%	98.93%	99.81%
	$APPT_{subtraction}$	97.31%	99.14%	95.29%	97.17%	99.46%
	$APPT_{product}$	98.82%	98.88%	98.69%	98.78%	99.78%
	$APPT_{CACHE}$	99.07%	**99.19%**	99.11%	99.05%	98.84%
	$APPT_{Tian\ et\ al.}$	99.06%	99.03%	**99.15%**	99.04%	98.85%
	$APPT_{concat}$	**99.13%**	99.09%	99.13%	**99.11%**	**99.86%**

TABLE 9: Effectiveness of APPT with different pre-trained models

Dataset	Model	Accuracy	Precision	Recall	F1-score	AUC
Small	$APPT_{bert}$	79.72%(↑ 4.32%)	80.84%(↑ 1.34%)	83.17%(↑ 6.67%)	81.76%(↑ 3.76%)	82.55%(↑ 2.25%)
	$APPT_{codebert}$	81.49%(↑ 6.09%)	82.10%(↑ 2.60%)	84.73%(↑ 8.23%)	83.35%(↑ 5.35%)	85.32%(↑ 5.02%)
	$APPT_{graphcodebert}$	81.83%(↑ 6.43%)	83.68%(↑ 4.18%)	83.63%(↑ 7.13%)	83.47%(↑ 5.47%)	85.79%(↑ 5.49%)
Large	$APPT_{bert}$	99.13%(↑ 0.53%)	99.09%(↑ 0.19%)	99.13%(↑ 0.93%)	99.11%(↑ 0.51%)	99.86%(↑ 0.96%)
	$APPT_{codebert}$	99.57%(↑ 0.97%)	99.71%(↑ 0.81%)	99.40%(↑ 1.20%)	99.55%(↑ 0.95%)	99.89%(↑ 0.99%)
	$APPT_{graphcodebert}$	99.61%(↑ 1.01%)	99.61%(↑ 0.71%)	99.59%(↑ 1.39%)	99.60%(↑ 1.00%)	99.90%(↑ 1.00%)

↑ denotes performance improvement against state-of-the-art technique CACHE.

dataset. This demonstrates the model choice may not impact the performance dramatically, and pre-trained models can consistently achieve state-of-the-art performance.

Specifically, to compare the performance of different pre-trained models, we can observe that both CodeBERT and GraphCodeBert achieve a better value for all metrics on the small dataset. This superior performance also generalizes to large datasets, where CodeBERT and GraphCodeBert have better or competitive (e.g., AUC) performance on the metrics. One possible explanation for this is that BERT is designed for natural language processing tasks, while CodeBERT and GraphCodeBERT regard a source code as a sequence of tokens or graph representation and then pre-train models on source code to support code-related tasks. This indicates that although pre-trained models in NLP can achieve state-of-the-art performance for assessing patch correctness, the adoption of pre-trained models targeting source code can further boost the improvement.

> **Answer to RQ2:** The performance under different choices demonstrates that: (1) all training components (e.g., fine-tuning and LSTM stack) positively contribute to APPT; (2) the concat of the buggy and patched vectors is better than other methods to distinguish the changed code snippets; (3) advanced pre-trained models can provide a stable even better performance.

5.3　RQ3: Cross-Project Prediction

Experimental Design. We have demonstrated that APPT achieves optimal performance in a cross-validation setting, which is the common practice in the APCA community. It is possible that the patches generated by different APR tools for the same bug are split into the training set and testing set. However, in a real-world scenario, it would be impossible to use the label from one patch to predict the other for the same bug. In this section, to further provide insights about APPT, we investigate the ability of APPT to predict patch correctness in a cross-project setting. In particular, we test all projects in Defects4J separately by creating training and testing sets per project. For example, when we test all patches from the project Chart, we use the patches from other projects as the training set to ensure the testing patches are new and unseen. Due to the space limitation, we choose CACHE [28] and Tian et al. [16] as the baselines, the former represents state-of-the-art and the latter is the most related work to our APPT.

Results. Table 10 presents the effectiveness APPT in a cross-project prediction scenario. The first column lists the name of the project. The second column lists the investigated APCA techniques. The third to seventh columns list the results of accuracy, precision, recall, F1-score and AUC. We also present the results by considering all five projects in the last row. The best results per project are shown in bold.

From the table, we can observe that APPT still substantially outperforms compared techniques by achieving 72.49% accuracy and 70.78% precision, i.e., 3.48% and 2.62% more than CACHE. Moreover, Recall and F1-score are consistently improved at least by 19.18% and 11.11% compared to CACHE. In addition, we can observe that compared to within-project prediction (i.e., RQ1), all techniques perform worse in the cross-project prediction scenario. For example, CACHE can predict 75.4% patches correctly while only 69.01% patches across different projects. As for APPT, it can predict 79.0% patches within projects while 72.49% patches across different projects. The observation is as expected, since in the within-project prediction scenario, testing data and training data may be from the same project, which tend to share similar features; whereas the cross-project prediction can be more challenging since characteristics between projects can be very different. Even though, we can

TABLE 10: Effectiveness of APPT in a cross-project setting

Project	Approach	Accuracy	Precision	Recall	F1-score	AUC
Chart	Tian et al.	58.90%	39.66%	41.82%	40.71%	55.64%
	CACHE	72.44%	**71.59%**	63.00%	67.02%	**78.94%**
	APPA	**73.49%**	71.01%	**76.44%**	73.61%	74.99%
Closure	Tian et al.	60.20%	61.64%	62.03%	61.83%	62.52%
	CACHE	64.00%	61.45%	51.00%	55.74%	**68.95%**
	APPA	**66.87%**	**69.29%**	**89.81%**	**78.23%**	63.18%
Lang	Tian et al.	63.37%	65.34%	62.78%	64.03%	68.16%
	CACHE	68.00%	66.28%	57.00%	61.29%	**75.76%**
	APPA	**72.98%**	**71.62%**	**71.00%**	**71.31%**	73.88%
Math	Tian et al.	58.22%	47.06%	49.72%	48.35%	57.03%
	CACHE	**69.78%**	69.62%	55.56%	61.80%	**76.33%**
	APPA	69.11%	**70.55%**	**84.25%**	**76.79%**	63.91%
Time	Tian et al.	71.11%	86.96%	66.67%	75.47%	75.33%
	CACHE	70.83%	**71.88%**	65.71%	68.66%	75.33%
	APPA	**80.00%**	71.43%	**66.67%**	68.97%	**83.11%**
All	Tian et al.	62.36%	60.13%	56.60%	58.31%	63.73%
	CACHE	69.01%	68.16%	58.45%	62.94%	**75.06%**
	APPA	**72.49%**	**70.78%**	**77.63%**	**74.05%**	71.81%

observe that compared to other techniques, APPT exhibits the smallest effectiveness drop between within-project and cross-project prediction. In summary, our results demonstrate that even when trained in the cross-project prediction scenario, APPT still consistently outperforms state-of-the-art APCA techniques on hundreds of extra bugs.

> **Answer to RQ3:** The performance under a cross-project scenario demonstrates that: (1) all investigated techniques show some decline in prediction performance compared with a cross-validation setting; (2) APPT still achieves optimal performance among four of five investigated metrics when predicting patch correctness from other projects.

6 DISCUSSION

6.1 Threats to Validity

To facilitate the replication and verification of our experiments, we have made the relevant materials (including source code, trained models, and patch data) available. Despite that, our study still faces some threats to validity, listed as follows.

The first threat to validity lies in the patch benchmark. We focus on the Defects4J database with reproducible real faults and collect 1,183 patches generated by existing APR tools. However, the patch benchmark may not consider all available APR tools. To address this, following the latest work [28], we include the 22 APR tools covering four repair categories. We also adopt the large benchmark containing 49,694 real-world patches and multiple evaluation metrics to evaluate the generalization ability of the studied techniques. Another potential issue is that the patch benchmark may contain mislabelled patches. In fact, these patches are manually labeled by the authors of the corresponding repair approaches, and then released for continuous review and correction by the APR community. There is also a re-checking process when they are collected for APCA research. Thus, we are confident in the reliability of these collected patches in our work, which have been employed

by most of previous patch correctness studies [16], [23], [64]. Overall, to the best of our knowledge, the used patch benchmarks are the largest set explored in the literature on patch correctness assessment.

The second threat to validity is that the performance of APPT may not generalize to other pre-trained models. We select BERT in our experiment due to its powerful performance in recent code-related works. However, it is unclear whether the conclusions in our experiment (discussed in Section 5) can be maintained when using other pre-trained models. We have mitigated the potential threat by using CodeBERT and GraphCodeBERT to demonstrate the performance of APPT under different pre-trained models. The investigated pre-trained models include both code-related ones (e.g., CodeBERT) and natural language-specific ones (e.g., BERT). We also rely on two diverse patch benchmarks to ensure the generality of the experimental conclusions.

The last threat to validity is the implementation of the baselines. In our work, we compare APPT against a wide range of APCA techniques with different categories. Implementing these baselines may introduce a potential threat to the internal validity. To mitigate this threat, following the recent work [28], we conduct the experiment under the same setting and reuse the released results from the original work [16], [17], [28]. Further, we carefully check the reused results and publicly release all our materials for further verification.

6.2 Comparison with BATS

In our work, following some recent APCA work [16], [17], 17 related APCA techniques with different categories (i.e., 2 representation learning-based ones, 9 dynamic-based ones, 4 static-based ones and 2 learning-based ones) are compared in our experiment (discussed in Section 5). To the best of our knowledge, the selected baselines are the largest set on patch correctness prediction in the literature. However, there may exist other possible techniques that could have been used. For example, the recent BATS [23] predicts patch correctness based on the similarity of failing test cases, which can be complementary to the state-of-the-art APCA techniques. We

TABLE 11: Comparison with a state-of-the-art learning-based APCA technique BATS

APCA	Threshold	#Patch	Accuracy	Precision	+Recall	-Recall	F1-score
BATS (0)	0	1278	52.50%	48.81%	62.82%	43.69%	54.94%
BATS (0.8)	0.8	114	67.54%	63.16%	84.21%	50.88%	72.18%
APPT		1278	**85.05%**	**83.39%**	**84.38%**	**85.67%**	**83.88%**

do not include BATS in our experiment (discussed in Section 5) because it requires historical test cases as the search space for searching similar cases, which are not available in our dataset.

We then perform an additional evaluation by assessing APPT on the dataset provided in BATS by the standard 5-fold cross-validation. However, BATS fails to assess some plausible patches as it considers only historical test cases with the similarity which are higher than a threshold. For example, BATS with 0.8 threshold value is able to predict only 8.9% (114/1278) of the plausible patches. Thus, we compare APPT against BATS with 0.0 threshold value, which can perform prediction for all patches. We also compare APPT against BATS with 0.8 threshold value, as it achieves the best recall, F1-score and AUC performance among all threshold values. The results are presented in Table 11. The first and second columns list APPT and BATS (with 0.0 and 0.8 threshold values, respectively). The third column lists the number of predicted patches under the same testing set. It is worth noting that BATS(0) and APPT can predict all patches (i.e., 1278 patches), while BATS(0.8) may fail to predict some plausible patches if no test case satisfies the threshold value. Each cell is represented as $x(y)$, where x is the number of patches predicted by APPT and BATS and y is the total number of patches in the dataset. The remaining columns list the detailed performance under the metrics. We also present the performance of +Recall and -Recall as BATS focuses on identifying both correct patches and overfitting patches.

From Table 11, we can find APPT achieves 83.39%~85.05%, improving the metrics by 21.56%~34.58% when compared with BATS (threshold is set to 0.0). When the threshold of BATS is set to 0.8, APPT can still improve the metrics by 12.40% on average while predicting 91.1% more plausible patches. Besides, APPT is able to achieve 84.38% for +Recall and 85.67% for -Recall, indicating its ability to predict both overfitting and correct patches. Overall, the results demonstrate that APPT performs better than BATS in terms of the number of predicted patches and the prediction metrics.

6.3 Correct Patch Identification

TABLE 12: Effectiveness of APPT in identifying correct patches

Approach	-Accuracy	-Precision	-Recall	-F1-score	-AUC
Tian et al.	64.84%	62.47%	55.70%	58.89%	55.64%
CACHE	76.33%	72.81%	76.07%	74.41%	74.99%
APPA	**79.12%**	**77.17%**	**76.45%**	**76.81%**	**78.94%**

In our work, following existing studies, we develop the pipeline of APPT to filter the overfitting patches, so as

to improve the precision of APR-generated patches. The experimental results show APPT achieves promising performance. To further evaluate APPT in practice, we employ APPT to identify the correct patches. We employ the same evaluation metrics as the overfitting patch filtering scenario (discussed in Section 4.5). It is worth noting that here the positive samples are correct patches and the negative samples are overfitting patches. For example, the recall measures to what extent correct patches are identified, i.e., the percentage of correct patches that are indeed predicted as correct. According to the definition of TP, FP, TN and FN, the Recall value is calculated by $-Recall = \frac{TN}{TN+FP}$. Similar to Section 6.3, we select the state-of-the-art technique CACHE [28] and the most related technique Tian et al. [16].

Table 12 presents the comparison results. We can find that APPT still achieves better performance with respect to all metrics when identifying the correct patches. For example, APPT reaches 79.02% accuracy, 77.17% precision, 76.45% recall, 76.81% F1-score, and 78.94% AUC, which are 2.79%, 4.36%, 0.37%, 2.40%, and 3.95% higher than the state-of-the-art CACHE. The improvement against Tian et al. reaches 14.29% ~ 23.30% on all metrics. Such results further reflect the promising results of APPT in the correct patch identification scenario.

7 IMPLICATION AND GUIDELINE

Based on the observations in our experiment, we can summarize the following essential practical guidelines for future patch correctness assessment studies.

Sequence features can work. Our study demonstrates that APPT, representing source code as a sequence of tokens, performs even better than structure-based features considering complex code-aware characteristics (e.g., abstract syntax tree in CACHE) and manually-designed features (e.g., ODS). Such observations indicate that features, such as code tokens, should not be just ignored and a systematic study to explore the impact of different code representations is needed in the future. In fact, they should be considered and even integrated with different features (e.g., data flow graph and abstract syntax tree) to design more advanced patch correctness assessment techniques.

The quality of the training dataset is important. We can find that APPT achieves 91.5% precision in Table 6 (involving 902 patches) while the precision is decreased by 10.8% in Table 4 (involving 1183 patches). Similar performance can also be observed in Lin et al. [28]. The results show that more training data cannot always lead to better performance for patch correctness assessment. It is crucial to automatically select the most informative training set that represents the whole patch benchmarks to optimize the prediction accuracy. For example, it is interesting to explore how the number of patches is distributed across fix

patterns and how to select balanced patches for each fix pattern. Future work can also be conducted to investigate training data selection approaches targeting specific bug benchmarks under prediction or even specific bug types under prediction.

Pre-trained model-based APCA techniques require more attention. Our results show that the BERT-based APPT performs even better than the state-of-the-art APCA techniques. Also, the CodeBERT-based and GraphCodeBERT-based APPT can further enhance the prediction effectiveness. Such observation motivates future researchers to investigate more advanced APCA techniques by employing different pre-trained models. It is interesting to conduct comprehensive and in-depth work to evaluate pre-trained models' capabilities (e.g., ChatGPT[5]) for the APCA task. Meanwhile, thorough evaluations are recommended to explore how different features, such as bug types and fix patterns, influence the performance of pre-trained models in patch correctness prediction.

8 RELATED WORK

In this paper, we adopt advanced pre-trained language models to address the important overfitting problem (i.e., the generated patches are overfitting to the given test suite) in automated program repair. Our work is related to automated program repair, patch correctness assessment and pre-trained models. We have introduced the existing work about patch correctness assessment in Section 4.3. Thus, in this section, we focus on and discuss the existing work on automated program repair techniques (Section 8.1) and pre-trained models (Section 8.2).

8.1 Automated Program Repair

APPT investigates patches generated by existing APR techniques in the literature [65]. These techniques are from different categories, i.e., heuristic-based [66], [67], constraint-based [68]–[70], template-based [71], [72] and learning-based repair techniques [39], [73], summarized as follows.

• *Heuristic-based repair techniques.* These techniques usually use a heuristic algorithm to find a valid patch by iteratively exploring a search space of syntactic program modifications [66], [67], [74]. Among them, GenProg [66] proposed in the early days has been considered a seminal work in this field, which uses genetic programming to search for correct repairs. GenProg represents candidate repairs as sequences of edits to source code and evaluate them by the execution results of test cases. Those candidates that pass more test cases are considered to have a higher fitness and are iteratively applied to produce new candidates based on mutation and crossover operations. GenProg is reported to fix 55 of 115 considered bugs, but only 2 of them are fixed correctly, raising the overfitting issue in the APR community. The recent SimFix technique [75] utilizes code change operations from existing patches across different projects and similar code snippets within the buggy project to build two search spaces. Then, the intersection of the above two search spaces is further used

to search the final patch using basic heuristics. SimFix is able to generate 34 correct patches along with 22 overfitting patches, leading to a precision of 60.7%.

• *Constraint-based repair techniques.* These techniques mainly focus on repairing conditional statements, which can repair more than half of the bugs repaired by existing APR approaches [68], [69], [76]. In detail, these techniques transform the patch generation into a constraint-solving problem, and use a solver to obtain a feasible solution. For example, Nopol [69] relies on an SMT solver to solve the condition synthesis problem and generates 5 correct patches among 35 plausible patches for Defects4J [77]. Furthermore, ACS [78] generates patches that are highly likely to be correct by refining the ranking of ingredients for condition synthesis and successfully generates 18 correct patches over 23 generated patches with a precision of 78.3%.

• *Template-based repair techniques.* These techniques generate patches by designing pre-defined fix patterns to mutate buggy code snippets with the retrieved donor code [71], [72], [79]. For example, Liu et al. [79] revisit the repair performance of repair patterns (i.e., Tbar) using a systematic study that evaluates the effectiveness of a variety of fix patterns summarized from the literature. Tbar is able to generate 101 plausible patches for 101 bugs and 74 of them are considered to be fixed correctly (i.e., semantically identical to the human-written patch). PraPR [80] is able to generate plausible patches for 148 real-world Defects4J bugs using JVM bytecode mutation but only 43 bugs are correctly fixed.

• *Learning-based repair techniques.* These techniques attempt to fix bugs enhanced by machine learning techniques [36], [39], [81]–[84] and are getting increasing attention recently. For example, Tufano et al. [82] extensively evaluate the ability of neural machine translation techniques to generate patches from bug-fixes commits in the wild. Chen et al. [47] propose a novel end-to-end approach (i.e., SEQUENCER) to program repair based on sequence-to-sequence learning. SEQUENCER generates plausible patches for 19 bugs and 14 bugs are considered to be correctly fixed. Li et al. [39] adopt a tree-based RNN encoder-decoder model (i.e., DLFix) to learn code contexts and transformations from previous bug fixes. Lutellier et al. [84] propose a new context-aware NMT architecture (i.e., CoCoNut) that represents the buggy source code and its surrounding context separately, to automatically fix bugs in multiple programming languages.

We find that although APR is widely investigated, the correctness of APR-generated patches is still a long-standing challenge in the literature [85]. In our experiment, we select 22 representative APR tools (e.g., SimFix, ACS, and SEQUENCER) from the four categories, representing state-of-the-art techniques in the corresponding category. Then we evaluate APPT on the plausible patches (i.e., passing the original test cases) generated by these APR techniques.

8.2 Pre-trained Model

Our approach is inspired by the application of pre-trained models in NLP and code-related tasks. In this section, we first introduce the existing studies about pre-trained models in NLP (Section 8.2.1) and SE (Section 8.2.2). We then discuss

---

5. ChatGPT. https://openai.com/blog/chatgpt, accessed March 2023

the application of pre-trained models to some code-related tasks in SE (Section 8.2.3).

8.2.1 Pre-trained Model in NLP

Recent work has demonstrated substantial gains on many NLP tasks and benchmarks by pre-training on a large corpus of text followed by fine-tuning on a specific task. For example, Devlin et al. [29] propose a new language representation model BERT to pre-train deep bidirectional representations from the unlabeled text by jointly conditioning on both left and right contexts in all layers. To explore the landscape of transfer learning techniques for NLP, Raffel et al. [31] propose a text-to-text transfer transformer T5 by introducing a unified framework that converts all text-based language problems into a text-to-text format. Brown et al. [30] propose an autoregressive language model GPT-3 without any gradient updates or fine-tuning, with tasks and few-shot demonstrations specified purely via text interaction with the model.

In this work, we choose BERT to encode a given plausible patch into a fixed-length representation vector as the input of the deep learning classifier, due to the powerful performance of BERT in previous work [86].

8.2.2 Pre-trained Model in SE

Inspired by the application of pre-trained models in NLP, many researchers apply the pre-trained model to code-related tasks. Instead of designing new network architectures, SE researchers usually adopt existing architectures in NLP and design some code-aware pre-training tasks (e.g., code-AST prediction and bimodal dual generation) to learn representations of the source code. Then the pre-trained models are further fine-tuned to some diversified code-related tasks such as code-code (clone detection, defect detection, cloze test, code completion, code refinement, and code-to-code translation), text-code (natural language code search, text-to-code generation), and code-text (code summarization) scenarios.

For example, Feng et al. [34] present a bimodal pre-trained model (*CodeBERT*) for natural language and programming languages by masked language modeling and replaced token detection to support code search and code documentation generation tasks. Guo et al. [35] present the first pre-trained model (*GraphCodeBERT*) that leverages code structure to learn code representation to improve code understanding tasks (i.e., code search, clone detection, code translation, and code refinement). Guo et al. [32] present UniXcoder, a unified cross-modal pre-trained model for programming language. UniXcoder utilizes mask attention matrices with prefix adapters to control the behavior of the model and leverages cross-modal contents such as AST and code comment to enhance code representation. In contrast to most studies pre-training a large-scale model from scratch costly, we attempt to boost patch correctness assessment on top of the existing pre-trained language model fine-tuning paradigm.

In this work, to further explore the generalization ability of APPT, we select other BERT-like models (i.e., CodeBERT and GraphCodeBERT) as the encoder stack due to their powerful performance in the code-related tasks.

8.2.3 Applications of Pre-trained Model in SE

In addition to the above-mentioned typical code-related tasks (e.g., automatic bug-fixing, injection of code mutants, generation of asserts in tests and code summarization in [87]), researchers have also applied pre-trained models to some other domains (e.g., code completion, and program repair [88]–[90]) in SE.

For example, Cinisell et al. [86] evaluate the performance of the BERT model in the task of code completion at different granularity levels, including single tokens, one or multiple entire statements. The results show that the model achieves promising results superior to state-of-the-art n-gram models, and the model learns better on some specific datasets (e.g., Android) when code abstraction is used. Ciborowska et al. [91] apply BERT to the bug localization problem with the goal of improved retrieval quality, especially on bug reports where straightforward textual similarity would not suffice. Recently, Salza et al. [92] investigate how transfer learning can be applied to code search by pre-training and fine-tuning a BERT-based model on combinations of natural language and source code. Mashhadi et al. [93] propose a novel pre-trained model-based APR technique by fine-tuning CodeBERT on the ManySStuBs4J benchmark and find the approach generates fix codes for different types of bugs with comparable effectiveness and efficacy compared with state-of-the-art APR techniques. Recently, Tian et al. [64] treat the APCA problem as a question-answering task and propose a learning-based approach that exploits a deep NLP model to classify the relatedness of a bug report with a patch description based on a pre-trained BERT model.

Although there exist some SE tasks (e.g., code review and bug localization) benefiting from pre-trained models, in this work, we perform the first application of fine-tuning pre-trained models to predict the generated patch correctness in automated program repair.

9 CONCLUSION

In this work, we present APPT, a novel pre-trained model-based automated patch correctness prediction technique by both pre-training and fine-tuning. We first adopt the off-the-shelf pre-trained model as the encoder stack and LSTM stack to enhance the dependency relationships among the buggy and patched code snippets. Then we build a deep learning classifier with two fully connected layers and a standard softmax function to predict whether the patch is overfitting or not. We conduct experiments on both patch datasets and show that APPT significantly outperforms state-of-the-art learning-based and traditional APCA techniques. We further demonstrate that APPT is generalizable to various pre-trained models. Based on these observations, some implications and guidelines on improving the existing learning-based techniques (e.g., the usage of simple features and pre-trained models) are provided. Overall, we highlight the direction of applying pre-trained models to predict patch correctness automatically.

ACKNOWLEDGMENTS

REFERENCES

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering (TSE'17)*, vol. 45, no. 1, pp. 34–67, 2017.

[2] S. Benton, X. Li, Y. Lou, and L. Zhang, "Evaluating and improving unified debugging," *IEEE Transactions on Software Engineering (TSE'21)*, no. 01, pp. 1–1, 2021.

[3] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE'09)*. IEEE, 2009, pp. 298–308.

[4] E. Winter, V. Nowack, D. Bowes, S. Counsell, T. Hall, S. Haraldsson, and J. Woodward, "Let's talk with developers, not about developers: A review of automatic program repair research," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 419–436, 2022.

[5] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM (CACM'19)*, vol. 62, no. 12, pp. 56–65, 2019.

[6] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. R. Winter, S. Counsell, D. Bowes *et al.*, "On the introduction of automatic program repair in bloomberg," *IEEE Software*, vol. 38, no. 4, pp. 43–51, 2021.

[7] E. R. Winter, V. Nowack, D. Bowes, S. Counsell, T. Hall, S. Haraldsson, J. Woodward, S. Kirbas, E. Windels, O. McBello *et al.*, "Towards developer-centered automatic program repair: findings from bloomberg," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1578–1588.

[8] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "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 (ESEC/FSE'19)*, 2019, pp. 302–313.

[9] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "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 ACM/IEEE International Conference on Software Engineering (ICSE'20)*, 2020, pp. 615–627.

[10] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE'19)*. IEEE, 2019, pp. 524–535.

[11] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–69, 2023.

[12] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE'16)*, 2016, pp. 702–713.

[13] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, 2014, pp. 64–74.

[14] Q. Zhang, Y. Zhao, W. Sun, C. Fang, Z. Wang, and L. Zhang, "Program repair: Automated vs. manual," *arXiv preprint arXiv:2203.05166*, 2022.

[15] Q. Zhang, X. Zhai, S. Xu, W. Huang, J. Zhang, and Y. Fan, "Interactive patch filtering via test generation," in *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2022, pp. 42–53.

[16] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*, 2020, pp. 981–992.

[17] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*, 2020, pp. 968–980.

[18] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Antipatterns in search-based program repair," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, 2016, pp. 727–738.

[19] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (FSE'11)*, 2011, pp. 416–419.

[20] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Proceedings of the Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA '07)*, 2007, pp. 815–816.

[21] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE'18)*, 2018, pp. 789–799.

[22] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering (TSE'22)*, vol. 48, no. 8, pp. 2920–2938, 2022.

[23] H. Tian, Y. Li, W. Pian, A. K. Kabore, K. Liu, A. Habib, J. Klein, and T. F. Bissyandé, "Predicting patch correctness based on the similarity of failing test cases," *ACM Transactions on Software Engineering and Methodology (TOSEM'22)*, vol. 31, no. 4, pp. 1–30, 2022.

[24] H. Tian, K. Liu, Y. Li, A. K. Kaboré, A. Koyuncu, A. Habib, L. Li, J. Wen, J. Klein, and T. F. Bissyandé, "The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches," *ACM Transactions on Software Engineering and Methodology*, 2022.

[25] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering (TSE'16)*, vol. 42, no. 8, pp. 707–740, Aug 2016.

[26] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*, 2020, pp. 75–87.

[27] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, 2015, pp. 532–543.

[28] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Transactions on Software Engineering and Methodology (TOSEM'22)*, vol. 31, no. 3, pp. 1–29, 2022.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pretraining of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[30] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," in *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS'20)*, vol. 33, 2020, pp. 1877–1901.

[31] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research (JMLR'20)*, vol. 21, pp. 1–67, 2020.

[32] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL'22)*, 2022, pp. 7212–7225.

[33] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, and Z. Chen, "A survey on large language models for software engineering," *arXiv preprint arXiv:2312.15223*, 2023.

[34] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics (EMNLP'20)*, 2020, pp. 1536–1547.

[35] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *Proceedings of the 9th Interna-

tional Conference on Learning Representations (ICLR'21), 2021, pp. 1–18.

[36] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21)*, 2021, pp. 1161–1173.

[37] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, 2018, pp. 1–15.

[38] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*, 2022, pp. 1169–1180.

[39] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*, 2020, pp. 602–614.

[40] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages (POPL'19)*, vol. 3, no. POPL, pp. 1–29, 2019.

[41] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR Poster'15)*, 2015, pp. 1–15.

[42] R. Just, D. Jalali, and M. D. Ernst, "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 (ISSTA'14)*, 2014, pp. 437–440.

[43] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, 2017, pp. 637–647.

[44] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th international conference on software engineering (ICSE'18)*, 2018, pp. 12–23.

[45] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE'18)*, 2018, pp. 1–11.

[46] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*, 2018, pp. 118–129.

[47] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering (TSE'19)*, vol. 47, no. 9, pp. 1943–1959, 2019.

[48] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*, 2020, pp. 573–577.

[49] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: a large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*, 2018, pp. 10–13.

[50] T. Durieux and M. Monperrus, "IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs," Universite Lille 1, Tech. Rep. hal-01272126, 2016.

[51] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An extensible java bug benchmark for automatic program repair studies," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, 2019, pp. 468–478.

[52] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "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 (SPLASH Companion'17)*, 2017, pp. 55–56.

[53] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)*, 2017, pp. 226–236.

[54] B. Yang and J. Yang, "Exploring the differences between plausible and correct patches at fine-grained level," in *Proceedings of the 2nd IEEE International Workshop on Intelligent Bug Fixing (IBF'20)*. IEEE, 2020, pp. 1–8.

[55] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, 2017, pp. 660–670.

[56] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax-and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 11th Joint Meeting on European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'17)*, 2017, pp. 593–604.

[57] X. Zhou, B. Xu, K. Kim, D. Han, T. Le-Cong, J. He, B. Le, and D. Lo, "Patchzero: Zero-shot automatic patch correctness assessment," *arXiv preprint arXiv:2303.00202*, 2023.

[58] T. Le-Cong, D.-M. Luong, X. B. D. Le, D. Lo, N.-H. Tran, B. Quang-Huy, and Q.-T. Huynh, "Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning," *arXiv preprint arXiv:2301.01113*, 2023.

[59] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *Empirical Software Engineering (ESE'21)*, vol. 26, no. 2, pp. 1–38, 2021.

[60] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, 2017, pp. 831–841.

[61] Y. Wei, Z. Quanjun, H. Tieke, F. Chunrong, H. Nguyen Quoc Viet, H. Xiaodong, and Y. Hongzhi, "Circle: Continual repair across programming languages," in *Proceedings of the 31th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*, 2022, pp. 427–438.

[62] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*, 2022, pp. 1506–1518.

[63] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*, 2020, pp. 518–529.

[64] H. Tian, X. Tang, A. Habib, S. Wang, K. Liu, X. Xia, J. Klein, and T. F. BissyandÉ, "Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness," in *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*, 2022, pp. 1–13.

[65] M. Monperrus, "The living review on automated program repair," HAL/archives-ouvertes.fr, Tech. Rep. hal-01956501, 2022.

[66] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering (TSE'12)*, vol. 38, no. 01, pp. 54–72, 2012.

[67] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016, pp. 441–444.

[68] T. Durieux and M. Monperrus, "Dynamoth: dynamic code synthesis for automatic program repair," in *Proceedings of the 11th International Workshop on Automation of Software Test (AST'16)*, 2016, pp. 85–91.

[69] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering (TSE'16)*, vol. 43, no. 1, pp. 34–55, 2016.

[70] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multi-line program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering (ICSE'16)*, 2016, pp. 691–701.

[71] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering (ESE'20)*, vol. 25, no. 3, pp. 1980–2024, 2020.

[72] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "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 (SANER'19)*, 2019, pp. 1–12.

[73] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program

repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*, 2021, pp. 341–353.

[74] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering (TSE'18)*, vol. 46, no. 10, pp. 1040–1067, 2018.

[75] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*, 2018, pp. 298–309.

[76] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'18)*. Springer, 2018, pp. 65–86.

[77] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering (ESE'17)*, vol. 22, pp. 1936–1964, 2017.

[78] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)*. IEEE, 2017, pp. 416–426.

[79] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*, 2019, pp. 31–42.

[80] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*, 2019, pp. 19–30.

[81] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, 2019, pp. 479–490.

[82] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM'19)*, vol. 28, no. 4, pp. 1–29, 2019.

[83] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*, 2017, pp. 1345–1351.

[84] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*, 2020, pp. 101–114.

[85] E. Winter, D. Bowes, S. Counsell, T. Hall, S. Haraldsson, V. Nowack, and J. Woodward, "How do developers really feel about bug fixing? directions for automatic program repair," *IEEE Transactions on Software Engineering*, 2022.

[86] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of bert models for code completion," in *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'21)*. IEEE, 2021, pp. 108–119.

[87] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21)*, 2021, pp. 336–347.

[88] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," *arXiv preprint arXiv:2310.08879*, 2023.

[89] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 535–547.

[90] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, "Pre-trained model-based automated software vulnerability repair: How far are we?" *IEEE Transactions on Dependable and Secure Computing*, 2023.

[91] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with bert," in *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*, 2022, pp. 946–957.

[92] P. Salza, C. Schwizer, J. Gu, and H. C. Gall, "On the effectiveness of transfer learning for code search," *IEEE Transactions on Software Engineering (TSE'22)*, vol. 1, no. 1, pp. 1–1, 2022.

[93] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *Proceedings Companion of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'21)*, 2021, pp. 505–509.