Neural Detection of Semantic Code Clones via Tree-Based Convolution

Hao Yu[1,2], Wing Lam[3], Long Chen[2], Ge Li[1,4]*, Tao Xie[1,3], and Qianxiang Wang[5]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
[2]School of Software and Microelectronics, Peking University, Beijing, China
[3]University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
[4]Institue of Software, EECS, Peking University, Beijing, China
[5]Huawei Technologies Co., Ltd, Beijing, China
{yh0315, clcmlxl, lige}@pku.edu.cn, {winglam2, taoxie}@illinois.edu, wangqianxiang@huawei.com

***Abstract*—Code clones are similar code fragments that share the same semantics but may differ syntactically to various degrees. Detecting code clones helps reduce the cost of software maintenance and prevent faults. Various approaches of detecting code clones have been proposed over the last two decades, but few of them can detect semantic clones, i.e., code clones with dissimilar syntax. Recent research has attempted to adopt deep learning for detecting code clones, such as using tree-based LSTM over Abstract Syntax Tree (AST). However, it does not fully leverage the structural information of code fragments, thereby limiting its clone-detection capability.**

To fully unleash the power of deep learning for detecting code clones, we propose a new approach that uses tree-based convolution to detect semantic clones, by capturing both the structural information of a code fragment from its AST and lexical information from code tokens. Additionally, our approach addresses the limitation that source code has an unlimited vocabulary of tokens and models, and thus exploiting lexical information from code tokens is often ineffective when dealing with unseen tokens. Particularly, we propose a new embedding technique called position-aware character embedding (PACE), which essentially treats any token as a position-weighted combination of character one-hot embeddings. Our experimental results show that our approach substantially outperforms an existing state-of-the-art approach with an increase of 0.42 and 0.15 in F1-score on two popular code-clone benchmarks (OJClone and BigCloneBench), respectively, while being more computationally efficient. Our experimental results also show that PACE enables our approach to be substantially more effective when code clones contain unseen tokens.

***Index Terms*—source code, semantic clone, clone detection, tree-based convolution, structural information, lexical information, embedding, token, AST, generalization**

I. Introduction

Clones of program source code (i.e., code clones) are a pair of similar code fragments that share the same semantics in terms of functionality but differ syntactically. Code clones generally result from developers' copy-and-paste behaviors, or by different developers implementing the same functionality. The existence of code clones is problematic for three main reasons. (1) Code clones unnecessarily increase program size. As a system increases in size, more software maintenance efforts are needed. (2) Changes to a code segment, such as fault fixing, need to be made to its clones as well, thereby increasing maintenance efforts. Also, if changes are performed inconsistently, faults could be introduced. (3) Duplicating a code fragment that contains faults leads to fault propagation. Although some may argue that code clones enable faster software development (i.e., developers are less reliant on the same piece of code), there is also common agreement that code clones should be detected and managed [1, 2].

One common taxonomy among researchers is to group code clones into four types [3, 4, 5]. The first three types (Type-1, Type-2, and Type-3) are largely concerned with textual similarities, while the remaining type (Type-4) is largely concerned with functional similarities. Type-4 clones are the most difficult to detect as they include clones that are highly dissimilar syntactically but still perform the same function. An example of a Type-4 clone is a code fragment implementing bubble sort and another implementing quick sort. To clarify the differences between Type-3 and Type-4 clones, researchers divide these two types into the following four categories based on their syntactical similarity (sorted from the easiest to most difficult to detect): Very-Strong Type-3, Strong Type-3, Moderately Type-3, and Weak Type-3/Type-4. In the rest of this paper, we refer to the two most-difficult-to-detect categories of clones as *semantic clones*.

Since the emergence of code clones as a research field, substantial efforts have focused on detecting and analyzing Type-1 to Type-3 clones, but they have had limited success with semantic clones such as Moderately Type-3 and Type-4 clones. For example, despite numerous research tools developed for detecting Type-3 clones, previous studies [6, 7] found that most Type-3-clone detectors do not perform as well as expected. Their findings of applying Type-3-clone detectors on a popular, public code-clone dataset, BigCloneBench [8], found that the existing state-of-the-art clone detectors achieved low recall on code clones with minor syntactic similarity.

In recent years, growing research efforts have adopted deep learning models to address various problems of programming languages and software engineering, such as code-clone detection. For example, Wei and Li [9] proposed to treat clone

*Corresponding author.

detection as a supervised-learning problem in their CDLH approach. The CDLH approach uses a Word2Vec [13] model to learn token embeddings to capture lexical information, and then trains an LSTM [10] model based on an Abstract Syntax Tree (AST) to combine these embeddings into a binary vector to represent a code fragment. The CDLH approach substantially surpasses previous approaches and is able to detect clones that are beyond the scope of traditional clone detectors.

However, the CDLH approach [9] does not make sufficient use of the structural information in an AST, despite such information being crucial to the semantics of the corresponding code. The CDLH approach neglects the information of AST node types and it only implicitly uses the structural information of the AST. We find that the clone-detection capability can be further improved by leveraging the information of the AST node types and using the structural information explicitly.

To fully unleash the power of deep learning for detecting code clones, we propose a new approach that uses tree-based convolution to detect semantic clones, by explicitly capturing both the structural information of a code fragment from its AST and lexical information from code tokens. In our approach, the tree-based convolution can capture subtree features in an AST. We further enhance the AST with code tokens by appending each token to the corresponding AST node as a new child node. In this way, we are able to make use of lexical information in addition to structural information.

Both our preceding basic approach and CDLH, along with some other studies [11], face a limitation of lacking generalization when training a clone detection model with code fragment pairs that are highly dissimilar to the code fragment pairs used to test the model. We refer to these highly dissimilar code fragment pairs as *unseen data*. These dissimilarities can be in the form of the vocabulary of the tokens and the semantic meaning of the pairs (e.g., all pairs used in training are related to list objects while all pairs used in testing are related to tree objects). To address the issue of unseen data, our approach includes a novel embedding technique called position-aware character embedding (PACE). Our PACE technique leverages and improves one-hot embedding [12] (previously proposed to embed words in natural language), which essentially converts every string to a number. The advantage of one-hot embedding is that words can be stored efficiently (since all it takes to store a word is one number). However, one major disadvantage of one-hot embedding is that it does not capture any information about similarity of words. To address this disadvantage, our PACE technique treats a token as a position-weighted combination of character one-hot embeddings. Compared to other embedding techniques such as Word2Vec [13], PACE does not use the semantic meaning of the word to detect similarity; instead, PACE relies on only the position and ordering of characters.

This paper makes the following main contributions:

Effective and efficient deep learning approach to detect semantic code clones. We propose to use tree-based convolution on token-enhanced ASTs to detect semantic code clones. Our approach does not need to transform an AST into a full binary tree, and can achieve higher effectiveness and efficiency than an existing state-of-the-art approach (CDLH) [9].

Novel technique to exploit lexical information in source code tokens. In our approach, we propose a novel embedding technique, PACE, to exploit lexical information in source code tokens and mitigate the threat that the unlimited vocabulary of tokens poses to generalization.

Experiments. We conduct experiments on two public code-clone datasets, and the experimental results show that our deep learning approach can substantially outperform CDLH by an increase of 0.42 and 0.15 in F1-score on two popular code-clone benchmarks (OJClone and BigCloneBench), respectively, while being more computationally efficient. Our experimental results also show that our PACE technique can make our approach more effective on unseen data while maintaining its effectiveness on seen data. Lastly, our experimental results on comparing multiple variations of our approach provide insights into how much structural or lexical information contributes to capturing the semantics of source code.

```
1  public void CopyFile(File source, File destination) {
2     fis = new FileInputStream(source);
3     fos = new FileOutputStream(destination);
4     byte[] buffer = new byte[4096];
5     int read;
6     while ((read = fis.read(buffer)) != -1) {
7        fos.write(buffer, 0, read);
8     }
9  }
10
11 public static void main(String[] args) {
12    InputStream in = null;
13    in = new URL(args[0]).openStream();
14    IOUtils.copyBytes(in, System.out, 4096, false);
15    IOUtils.closeStream(in);
16 }
```

Fig. 1. Example of a Type-4 clone.

II. Background

Code clones may differ syntactically to various degrees. One common taxonomy is to group code clones into the following four types [3, 4]:

- Type-1 (T1): syntactically identical code fragments, except for differences in white space and comments.
- Type-2 (T2): in addition to Type-1 clone differences, syntactically identical code fragments, except for differences in identifier names and literal values.
- Type-3 (T3): in addition to Type-2 clone differences, syntactically similar code fragments that differ at the statement level. These fragments can have statements added, modified, or removed with respect to each other.
- Type-4 (T4): syntactically *dissimilar* code fragments that are still the same semantically. For example, one code fragment implementing bubble sort and another code fragment implementing quick sort are considered a pair of Type-4 clones.

Figure 1 shows an example of a Type-4 clone from BigCloneBench [8]. The two methods in the figure are both

used to copy a file. We can see that the code fragments that implement the same functionality have great differences in their structures and tokens. In this case, SourcererCC [7], which uses only token information, and Deckard [14], which uses only structural information, fail to detect this code clone. However, deep learning approaches can help learn the necessary information to detect this code clone. During training, deep learning approaches can learn different implementations of file copying. For example, CDLH [9] may be able to learn the semantic information of file copying from many code fragments during training, and therefore can detect this code clone during testing. However, one major limitation of CDLH is the information loss when the code fragments are too large. Section IV provides detailed explanations for why our approach can detect code clones such as the one in Figure 1 and even larger ones that can be challenging for CDLH.

III. Problem Definition

Given two code fragments C_i and C_j, we set their label $y_{i,j}$ to 1 if (C_i, C_j) is a clone pair or -1 otherwise. Then a set of training data of n code fragments $\{C_1, ..., C_n\}$ can be represented as $D = \{(C_i, C_j, y_{i,j})|i, j \in n, i < j\}$. Our goal is to train a deep learning model to learn a function ϕ that maps any code fragment C to a feature vector $\boldsymbol{v}$ so that for any pair of code fragments (C_i, C_j), the cosine similarity $s_{i,j}$ of the two feature vectors v_i and v_j is as close to the corresponding label $y_{i,j}$. We use Equation 1 to calculate the cosine similarity of two vectors of the same dimension:

$$Cosin\ Similarity(u, v) = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|} \tag{1}$$

Thus, we have the following equation (Equation 2):

$$s_{i,j} = \frac{\phi(C_i) \cdot \phi(C_j)}{\|\phi(C_i)\|\|\phi(C_j)\|} \tag{2}$$

where $s_{i,j} \in [-1, 1]$.

To determine whether a pair of code fragments (C_i, C_j) is a clone pair or not during inference, we need to set a threshold value σ such that (C_i, C_j) is a clone pair if $s_{i,j} \geq \sigma$. We choose σ empirically based on the validation set.

IV. Proposed Approach

In this section, we first introduce the overview of our proposed approach based on tree-based convolution, originally proposed to solve a program-classification problem [15, 16] for clone detection. We next explain the technical details of tree-based convolution and "continuous binary tree." We then use an example to illustrate how we enhance an AST with tokens in the corresponding code fragment. Finally, we explain the motivation and implementation detail of our novel token embedding technique, PACE.

A. Approach Overview

Figure 2 shows the overview of our approach named Tree-Based Convolution for Clone Detection (TBCCD). To process a code fragment, we first parse it into its AST. We then produce an additional AST, denoted as AST+, resulted from enhancing the AST with tokens in the code fragment. The users of our approach can configure it to use either AST or AST+ to conduct clone detection. Section VI describes experimental results using variants of our approach with different configurations.

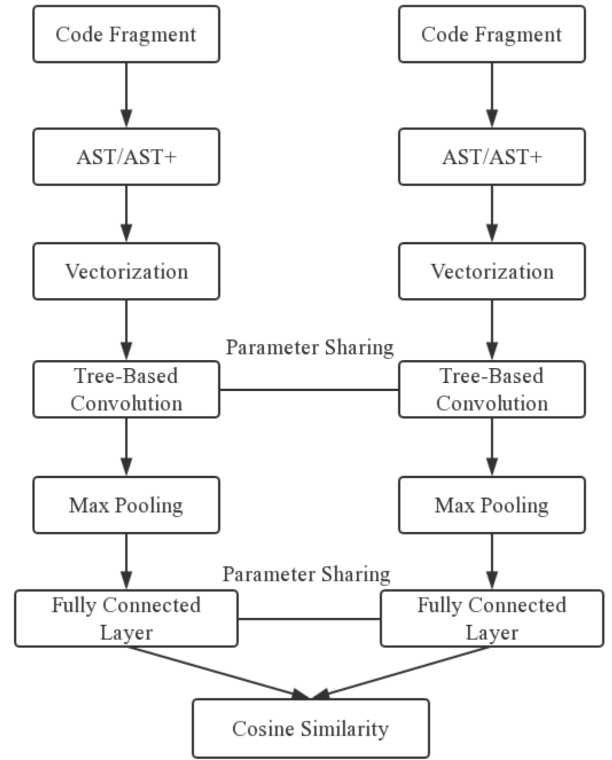

Fig. 2. The overview of TBCCD (AST+ denotes token-enhanced AST).

Then we initialize the embeddings of each type of AST nodes (and tokens in the case of AST+), before feeding the vectorized AST/AST+ into tree-based CNNs with a max pooling layer and a fully connected layer. To detect code clones, we use two neural networks in parallel to process a pair of code fragments at the same time. The two neural networks share the same set of parameters. We take the output of the fully connected layer as the feature vector of a code fragment and calculate the cosine similarity of the two vectors. Then we train the neural networks through gradient descent backpropagation to minimize the following loss function:

$$\sum_i \sum_j (s_{i,j} - y_{i,j})^2 \tag{3}$$

In summary, TBCCD learns to make the cosine similarity of non-clone pairs as close to -1 as possible and the cosine similarity of clone pairs as close to 1 as possible.

B. Tree-Based Convolution and Max Pooling

The most common convolution kernels have the shape of a square, whereas in tree-based convolution, each kernel (i.e., feature detector) takes the shape of a triangle, as illustrated in Figure 3. A set of fixed-depth feature detectors ($d \geq 2$, where d denotes the depth) are used to slide over the entire tree to capture subtree features. Each node in the AST/AST+ is the uppermost node in the sliding window for exactly once. When

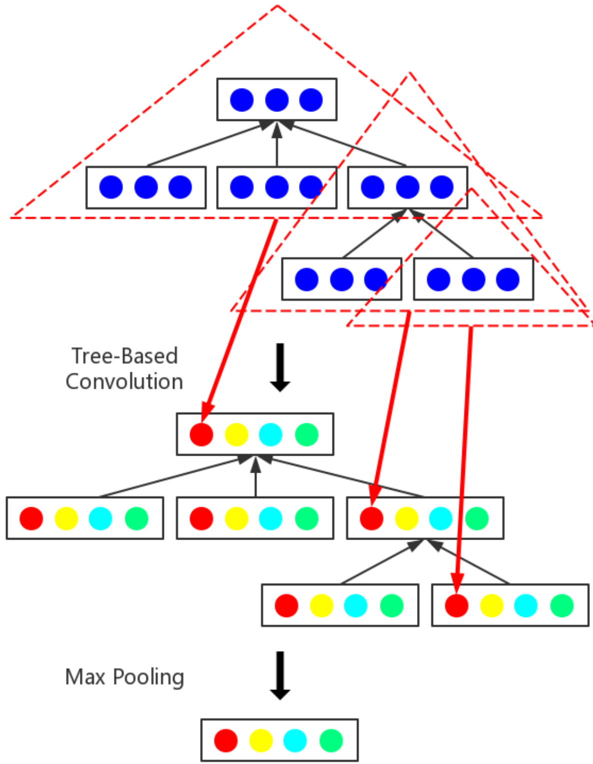

Fig. 3. Illustration of tree-based convolution with max pooling. In this case, there are four feature detectors, each with a fixed depth of two. The step of tree-based convolution is illustrated with only one feature detector using the red color. The tree structure maintains the same shape before and after the convolution, while the dimension of each node's vector has changed from the original three to four, which is the number of feature detectors. After tree-based convolution, max pooling is applied to each dimension of the vectors.

there are not enough layers of nodes in the sliding window, we add vectors of all zeros to represent the missing nodes. After tree-based convolution, max pooling is applied to each dimension in the output vectors; each dimension corresponds to one feature detector, thereby reducing an AST/AST+ of any depth to a fixed-length vector.

Formally, in a fixed-depth window of d, if there are n nodes with vector representations $x_1, ..., x_n$, where $x_i \in \mathbb{R}^{N_f}$, N_f is the dimension of the vector representations, then the output of tree-based convolution is

$$y = tanh(\sum_{i=1}^{n} W_{conv,i} \cdot x_i + b_{conv}) \quad (4)$$

where $y, b_{conv} \in \mathbb{R}^{N_f}, W_{conv,i} \in \mathbb{R}^{N_c \times N_f}$ (N_c is the number of feature detectors).

C. Continuous Binary Tree

One problem of the tree-based convolution is that unlike the nodes in a parse tree of natural languages whose number of children is limited to two, an AST node theoretically could have an unlimited number of children (e.g., the case statement in a switch expression). Such unlimited number is problematic because the number of weight matrices, i.e., $W_{conv,i}$ in Equation 4, cannot be determined. A commonly used workaround [11, 9] is to convert an AST into a full

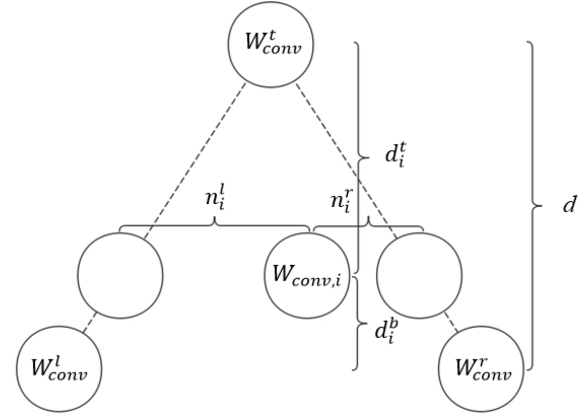

Fig. 4. Illustration of adopting "continuous binary tree" in tree-based convolution.

binary tree according to a pre-defined set of rules. However, some structural information could be lost during the process. We propose a different technique where any subtree in an AST is viewed as a "binary" tree, regardless of its size and shape. Such an AST is referred to as a "continuous binary tree" [15, 16].

As demonstrated in Figure 4, there are only three weight matrices in tree-based convolution: W^t_{conv}, W^l_{conv}, and W^r_{conv} where superscripts t, l, r refer to "top", "left", and "right", respectively. For node x_i in a sliding window, its weight matrix for convolution $W_{conv,i}$ is a linear combination of W^t_{conv}, W^l_{conv}, and W^r_{conv}, whose coefficients are computed according to the relative position of a node in the sliding window. d^t_i denotes the number of layers above, d^b_i denotes the number of layers below, n^l_i denotes the number of nodes to the left (on the same layer), and n^r_i denotes the number of nodes to the right. Thus we can define $W_{conv,i}$ as

$$W_{conv,i} = \frac{d^b_i}{d^b_i + d^t_i} W^t_{conv} + \frac{d^t_i}{d^b_i + d^t_i} W^b_{conv,i} \quad (5)$$

where $d^b_i + d^t_i = d$, and we have defined an intermediate variable $W^b_{conv,i}$, which is written as:

$$W^b_{conv,i} = \begin{cases} \frac{n^r_i}{n^r_i + n^l_i} W^l_{conv} + \frac{n^l_i}{n^r_i + n^l_i} W^r_{conv} & n^l_i \geq 1 \; or \; n^r_i \geq 1, \\ \frac{1}{2} W^l_{conv} + \frac{1}{2} W^r_{conv} & n^l_i = n^r_i = 0. \end{cases} \quad (6)$$

In our experiments, we choose $d = 2$, which we find to be sufficient.

D. Token-Enhanced AST (AST+)

Tokens in source code generally include keywords, constants, identifiers, strings, special symbols, and operators. Statistics show that the majority of code tokens are user-defined identifiers. The names of these identifiers would carry a lot of semantic information if developers follow certain naming conventions; such semantic information could supplement the structural information in an AST.

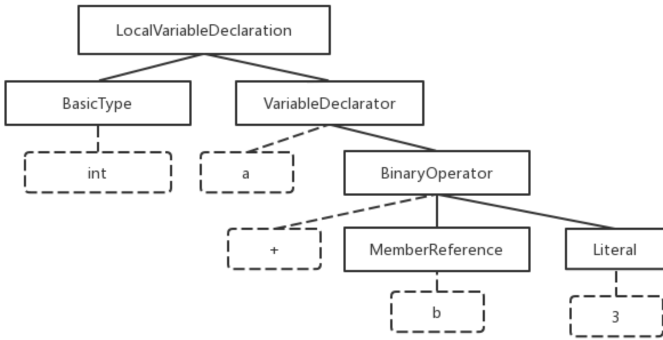

Fig. 5. Illustration of a token-enhanced AST for the Java code snippet `int a = b + 3`.

Previous approaches [11, 9] have also used the lexical information in tokens in their models, using variants of RNN. In our approach, we design a novel way to incorporate lexical information into tree-based CNN to further improve the ability of our approach to detect semantic clones. Specifically, we add each token in the code fragment as a child node to the corresponding AST node in the original AST and assign to the child node a vector of the same dimension as the AST node's vector. Then we apply the same tree-based convolution over the token-enhanced AST. To initialize the embeddings of the tokens, we treat each code fragment as a sequence of tokens and use Word2Vec [13] to learn the token embeddings.

Figure 5 shows an example of turning a Java code snippet into a token-enhanced AST. The tokens that we add to the AST is identified with dashed-line boxes, and the AST (in solid lines) represents a simple line of Java code, `int a = b + 3`. As shown in Section VI, the additional lexical information from tokens substantially improves the F1-score for clone detection, especially on BigCloneBench.

E. Position-Aware Character Embedding (PACE)

The vocabulary size of tokens is a special challenge to modeling source code [17]. In natural language processing (NLP), approaches usually limit vocabulary to the most common words, e.g., top 30,000 words during data processing. The out-of-vocabulary tokens are replaced by a special "⟨UNK⟩" token. Doing so would not always work well since there is an unlimited number of code tokens (e.g., developers can name identifiers in arbitrary ways). If we keep a regular vocabulary size for source code, there would be too many "⟨UNK⟩" tokens. Hellendoorn and Devanbu [17] have shown that it is unreasonable for source code models to use such a vocabulary.

This challenge poses a dilemma where one has to increase the vocabulary size of tokens to avoid too many "⟨UNK⟩" tokens for the model to perform well, but doing so makes it more difficult to train the deep neural networks. Even with a very large vocabulary size of tokens, a model still may not generalize to source code that uses out-of-vocabulary tokens.

To address this problem, we propose a novel embedding technique called position-aware character embedding (PACE), which essentially treats each token in source code as a position-weighted combination of character one-hot embeddings. First, we gather all possible characters in the source code tokens (except strings) and determine the total number n, which is the dimension of token embeddings. Then we transform the set of unique characters into one-hot embeddings. For a token that has k characters denoted as $c_1, c_2, ..., c_k$, its embeddings can be obtained with the following equation:

$$\sum_{i=1}^{k} \frac{k-i+1}{k} \times emb[c_i] \tag{7}$$

where $emb[c_i]$ is the one-hot embedding of c_i.

The motivation for PACE stems from our observation that the names of similar identifiers often appear to be similar. For example, in the Java JDK, there are classes such as LinkedList and ArrayList, which are ordered collections that allow duplicates. One does not need to train a language model to determine that these names share similar semantics; one can simply tell that they are similar because they share common characters and the characters appear in a similar order. There is good chance that the names of the instances of these classes may appear to be similar as well. Thus, our proposed PACE technique essentially encodes a token as a position-weighted sum of its character one-hot embeddings.

V. Experimental Setup

We evaluate the effectiveness of our approach by addressing the following research questions:

RQ1: How effective are the different variants of our approach at detecting code clones?

RQ2: How does the use of structural and lexical information of source code affect the effectiveness of detecting code clones?

RQ3: How does our approach's TBCCD+token variant compare to existing state-of-the-art approaches at detecting code clones?

RQ4: How effective is our approach at detecting code clones when the training set and test set consist of code fragment pairs with different semantic meanings?

We address RQ1 to understand which variants of our approach are most effective at detecting code clones, where each variant uses more or less information (e.g., type of the AST nodes). We then address RQ2 to provide insights into why certain variants can be more or less effective, and RQ3 to evaluate how effective and efficient our approach is when compared to the existing state-of-the-art approaches. Lastly, we evaluate our approach's clone detection ability when the training set and test set consist of code fragments with different semantic meanings.

A. Datasets

Our experiments use BigCloneBench [8] and OJClone [15], two public datasets commonly used to evaluate approaches of detecting code clones.

BigCloneBench, released by Svajlenko et al. [8], is the first big-data-curated benchmark of real clones to evaluate modern tools of detecting code clones. It was built by mining clones of frequently implemented functionalities from 25,000 Java systems (totaling 365 million lines of code), and the clones

TABLE I
PERCENTAGE OF CLONE TYPES IN BIGCLONEBENCH (T1: TYPE-1; T2: TYPE 2; VST3: VERY-STRONG TYPE-3; ST3: STRONG TYPE-3; MT3: MODERATELY TYPE-3; WT3/T4: WEAK TYPE-3/TYPE-4).

Clone Type	T1	T2	VST3	ST3	MT3	WT3/T4
Percent (%)	0.455	0.058	0.053	0.19	1.014	98.23

are verified by three domain experts. The current version of BigCloneBench has about 8 million tagged true clone pairs covering 43 functionalities [6]. The authors of some recent approaches, e.g., [18, 9], still chose to use the first version of BigCloneBench [8], which contains only 6 million tagged true clone pairs and 260,000 tagged false clone pairs, covering 10 functionalities. To compare our approach with these approaches, we also use that version of BigCloneBench.

Due to the ambiguity between the definitions of Type-3 and Type-4 clones, the creators of BigCloneBench further divided these two types of clones into four categories based on their syntactical similarity: Very-Strong Type-3 with [0.9, 1) similarity, Strong Type-3 with [0.7, 0.9) similarity, Moderately Type-3 with [0.5, 0.7) similarity, and Weak Type-3/Type-4 with [0, 0.5) similarity. The syntactical similarity is measured as the ratio of lines or tokens that a code fragment shares with another after Type-1 and Type-2 normalization, identified by the Linux tool "diff". Table I summarizes the data distribution in the first version of BigCloneBench in terms of clone types. Since the majority of code clone pairs are Weak Type-3/Type-4 clones, BigCloneBench is quite appropriate to be used for evaluating the detection of semantic clones.

OJClone, released by Mou et al. [15], is another public dataset used to evaluate code-clone detection. Originally the dataset was used for program classification but it has since been used by others [9] for code-clone detection as well. The dataset consists of solutions submitted by students to 104 programming questions on OpenJudge [19], written in C. For each question, there are 500 corresponding solutions, each of which is verified to be correct by OpenJudge. Solutions for the same question are considered clones.

B. Experimental Setting

1) RQ1 and RQ2: We conduct experiments on both datasets using the 1:1:8 experimental setting for validation, test, and training sets, respectively. We evaluate our approach in this experimental setting since it is commonly used in other related work [20, 21, 22]. For the BigCloneBench dataset, we use the same 9,134 code fragments from related work [9]. For the OJClone dataset, we select 500 solutions for each of the first 15 questions (question IDs 1-15), amounting to $15 \times 500 = 7,500$ code fragments.

2) RQ3: Our goal is to evaluate our proposed approach against CDLH [9], the existing state-of-the-art approach that uses AST-based LSTM to detect semantic clones. Thus, we strive to replicate CDLH's experimental setting as closely as possible for both datasets.

For the BigCloneBench dataset, the code fragments that we use in RQ1 are the same as the ones used by the CDLH authors [9]. Just as the authors of CDLH do for their experiments, we randomly select 500 code fragments for the validation set, 500 code fragments for the test set, and used the remaining 8,134 code fragments for the training set. We have 124,750 pairs of code fragments in the validation set and test set where true clone pairs account for 13.9% and 14.0%, respectively, and 33,068,778 pairs of code fragments in the training set. Lastly, we randomly sample about one million pairs from these pairs where true clone pairs account for 14.2%.

For the OJClone dataset, the 7,500 code fragments that we use in RQ1 are the same as the ones used by the CDLH authors in their experiments. Just as the authors of CDLH do for their experiments, we randomly select 500 code fragments for the validation set, 500 code fragments for the test set, and used the remaining 6,500 code fragments for the training set. To construct clone pairs, we pair every two code fragments together in the validation, test, and training sets. Thus, we have 124,750 pairs of code fragments in the validation set and test set where true clone pairs account for 6.7% and 6.8%, respectively, and 21,121,750 pairs of code fragments in the training set. Due to the vast number of pairs in the training set, we randomly sample about one million pairs from these pairs where true clone pairs account for 6.6%.

Our correspondence with the CDLH authors inform us of only the proportion that they use for the validation, test, and training sets. Therefore, due to the randomness in their selection of code fragments, we are unable to use the exact same set of code fragments in our experiments. To mitigate any bias from randomly selecting our validation, test, and training sets, we report the medians of results (precision, recall, and F1-score) with the median F1-score from randomly generating the sets ten times in Section VI-C. We also report the averages for precision, recall, and F1-score individually.

3) RQ4: Our goal is to evaluate our approach's generalization ability by using code fragment pairs that are substantially different semantically when conducting training and testing with our approach. In contrast to the 1:1:8 experimental setting used for RQ1, which at times have pairs in the training set that are semantically similar to pairs in the test set, our experimental setting for RQ4 ensures that all pairs in the training set are semantically different than the pairs in the test sets.

To accomplish this goal, we use the OJClone dataset, which contains 104 questions. Similar to RQ1, we use the first 15 questions (question IDs 1-15) and the 500 solutions per question to obtain 7,500 code fragments. From these code fragments, we then use the same 6,000 as what we use for the training set in RQ1. For the test sets, we create in total six sets each composed of the solution code fragments from the 15 questions. The first test set uses question IDs 16-30, the second test set uses question IDs 31-45, and so on. The last test set uses question IDs 90-104 and question ID 16. For each test set, we randomly select 750 code fragments from the possible 7,500 solutions. We do not use any code fragments

TABLE II
EFFECTIVENESS OF OUR APPROACH'S VARIANTS.

Approach	BigCloneBench P	BigCloneBench R	BigCloneBench F1	OJClone P	OJClone R	OJClone F1
TBCCD	0.78	0.73	0.76	0.97	0.95	0.96
TBCCD+token	0.95	0.95	**0.95**	0.99	0.99	**0.99**
TBCCD+token−type	0.94	0.95	**0.95**	0.92	0.85	0.88
TBCCD+token+PACE	0.94	0.96	**0.95**	0.99	0.99	**0.99**

TABLE III
INFORMATION SOURCES USED BY DIFFERENT APPROACHES.

Approach	AST	AST Node Type	Code Token
CDLH	✓	×	✓
TBCCD	✓	✓	×
TBCCD+token	✓	✓	✓
TBCCD+token−type	✓	×	✓
TBCCD+token+PACE	✓	✓	✓

for validation and instead simply use the same threshold as the one from RQ1's validation set.

4) All RQs: We choose the other hyper-parameters for training our approach as follows: the number of convolution kernels is 600 and the depth of sliding window is 2; the dimension of the fully connected layer is 50. We train the model for ten epochs and use the SGD optimizer with batch size 1 to train the hyper-parameters. The threshold of our approach for prediction is determined with the validation set of RQ1-RQ3. The threshold for RQ4 is the same as the one that we use for RQ1. Lastly, we use TensorFlow [24] to implement our models and our code is publicly available. The datasets that we use for each RQ are also publicly available [23].

VI. EXPERIMENTAL RESULTS

A. RQ1: Effectiveness of our approach's different variants

Our approach evaluated in the experiments includes four variants:

- TBCCD: tree-based convolution over AST. The dimension of AST node embedding is 100. We randomly initialize the embedding for each AST node type and jointly learn the embeddings of AST node and subtree features at the same time during the training.
- TBCCD+token: tree-based convolution over token-enhanced AST (AST+). The dimension of both AST node embedding and token embedding is 100. Token embeddings and AST node embeddings are randomly initialized and learned during training.
- TBCCD+token−type: tree-based convolution over AST+ but information regarding the AST nodes' type is omitted, where each AST node type has the same embedding. The token embeddings are the same as those in TBCCD+token.
- TBCCD+token+PACE: tree-based convolution over AST+, using our PACE embedding technique for both AST nodes and tokens. The dimension of the embeddings is dependent on the number of unique characters in the dataset (being 78 for BigCloneBench and 79 for OJClone). We treat the lower case and upper case of the same letter as two different characters, since capitalization may convey additional meaning.

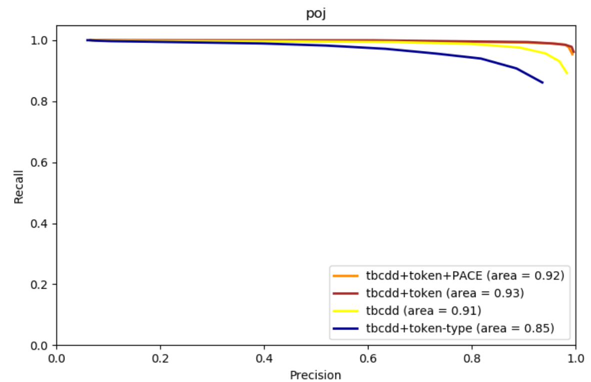

Fig. 6. The PR-curve of our approach's four variants on the validation set of OJClone.

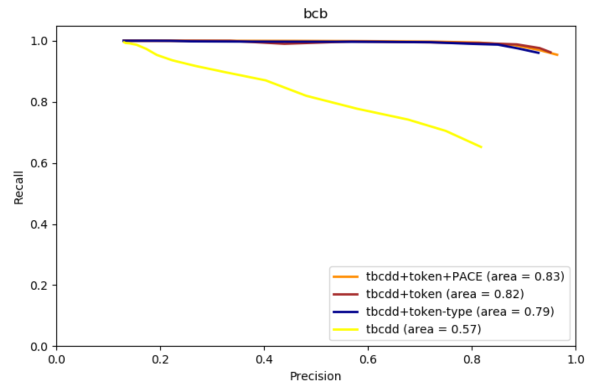

Fig. 7. The PR-curve of our approach's four variants on the validation set of BigCloneBench.

For each variant of our approach, we choose the set of parameters that yield the best F1-score on the validation set to evaluate its performance on the test set. Table II summarizes the experimental results and Table III lists the information sources of each variant along with related work CDLH [9].

As shown in Table II, after we incorporate our PACE technique with TBCCD+token, its F1-score barely changes. It seems that PACE, our new embedding technique, does not affect the effectiveness much. Yet as discussed in Section VI-D, TBCCD+token+PACE has been shown to be the most effective variant when the semantic meaning of the training and test sets dramatically differ.

We further draw the PR-curve of our approach's each variant to compare their AUCs, as shown in Figures 6 and 7. We vary the threshold σ defined in Section III and record the precision and recall of each variant on the validation set. Similar to Table II, these two figures show that *TBCCD+token* and *TBCCD+token+PACE* are the best variants on the two benchmark datasets. TBCCD+token−type has a drop in AUC on OJClone compared to the others, while TBCCD has the worst AUC on BigCloneBench.

TABLE IV
TOTAL AND AVERAGE OCCURRENCES OF THE MOST FREQUENT AST NODE TYPES IN BIGCLONEBENCH.

Rank	AST Node Type	Total	Average
1	MethodInvocation	182,011	19.9
2	MemberReference	168,330	18.4
3	Literal	130,885	14.3
4	ReferenceType	115,807	12.7
5	StatementExpression	107,205	11.7
6	BinaryOperation	68,019	7.4
7	VariableDeclarator	63,947	7.0
8	LocalVariableDeclaration	59,732	6.5
9	ClassCreator	35,966	3.9
10	Assignment	32,160	3.5
11	BlockStatement	29,902	3.3
12	IfStatement	24,878	2.7

TABLE V
TOTAL AND AVERAGE OCCURRENCES OF THE MOST FREQUENT AST NODE TYPES IN OJCLONE.

Rank	AST Node Type	Total	Average
1	ID	421,381	56.2
2	Constant	159,853	21.3
3	BinaryOp	129,421	17.3
4	TypeDecl	84,896	11.3
5	Decl	82,736	11.0
6	IdentifierType	79,422	10.6
7	ArrayRef	68,035	9.1
8	Assignment	67,163	9.0
9	UnaryOp	54,689	7.3
10	Compound	49,471	6.6
11	FuncCall	35,301	4.7
12	ExprList	35,152	4.7

B. RQ2: Effectiveness of structural and lexical information

Comparing the effectiveness of TBCCD+token, TBCCD+token−type, and TBCCD as shown in Table II, we have the following findings. (1) Structural information (AST Node Type) and lexical information (Code Token) are complementary to each other. When combined together, TBCCD+token achieves the highest F1-score compared to TBCCD+token−type and TBCCD. (2) Structural information matters more in OJClone than BigCloneBench, since without the information of AST node type, TBCCD+token−type drops dramatically from 0.99 to 0.88 in F1-score on OJClone compared to TBCCD+token, but remains the same 0.95 in F1-score on BigCloneBench. (3) Lexical information matters more in BigCloneBench, since without the information from source code tokens, TBCCD drops dramatically from 0.95 to 0.76 in F1-score on BigCloneBench compared to TBCCD+token, but only slightly from 0.99 to 0.96 in F1-score on OJClone. To better understand the dramatic drop in F1-score on BigCloneBench but not on OJClone, we study the the frequency of each AST node type. Tables IV and V summarize the frequent AST node types in the two datasets.

In BigCloneBench, the most frequent AST node type is "MethodInvocation," which appears 182,011 times across 9,134 code fragments, yielding an average of 19.9 per code fragment. On the other hand, OJClone's "FuncCall" node type ranks the 11th in terms of frequency, appearing 35,301 times across 7,500 solutions, an average of 4.7 per solution. One drawback of TBCCD is that the vanilla AST that it uses regards every method invocation as the same. The semantics of the functionality of the called method is lost. We believe that this loss of semantics and the discrepancy of method invocation frequencies between the two datasets cause the dramatic drop in effectiveness of TBCCD on BigCloneBench yet only a slight drop on OJClone. We suspect that the difference in the distribution of AST node types may be attributed to the programming environment for the dataset. The data from OJClone come from a programming environment where the users write up solutions from scratch, whereas BigCloneBench is mined from large open-source repositories where the usage of libraries is much more frequent.

C. RQ3: Effectiveness of our approach compared to state-of-the-art approaches

Apart from CDLH, we also compare our approach against three other related approaches in Table VI (as evaluated by the CDLH authors [9]):

- Deckard [14]: a popular AST-based approach that also uses a feature vector to represent an AST and its subtrees.
- Deep Learning for Code Clones (DLC) [11]: a recent approach that explores the use of deep learning for detecting code clones. It uses a recursive autoencoder to extract unsupervised deep features.
- SourcererCC [7]: a state-of-the-art token-based approach. It can detect some Type-3 clones despite using a bag-of-tokens representation.

Note that the preceding three related approaches do not need to be trained in a supervised manner, and thus their results should not vary much on different test sets from the same dataset.

We use the same experimental settings as what the CDLH authors use for their experiments to evaluate our approach against the other related approaches in terms of precision (P), recall (R), and F1-score (F1). As shown in Table VI, our approach outperforms CDLH in terms of F1-score on both datasets. By exploiting both the structural information in the AST and lexical information in code tokens, TBCCD+token achieves an increase of 0.42 in F1-score on OJClone and an increase of 0.15 in F1-score on BigCloneBench, compared to CDLH.

CDLH neglects the information of AST node types and uses only the structure of ASTs as a guidance for the order to encode each token in a sequence into one vector at the root; therefore, it does not fully utilize the structural information in source code. On the contrary, tree-based CNN is specifically designed to capture the structural information of programming languages; the structural information is essential to the semantics of the code. While CDLH is intended to capture both lexical and syntactical information, as suggested by the title of the paper [9], our approach outperforms CDLH on OJClone using only syntactical information, as shown in Table VI. Our results show that CDLH does not sufficiently capture

TABLE VI
EFFECTIVENESS OF OUR APPROACH AND OTHER RELATED APPROACHES USING THE SAME EXPERIMENTAL SETTINGS AS THOSE USED BY [9]. RESULTS OF APPROACHES MARKED WITH ‡ ARE TAKEN DIRECTLY FROM [9].

	BigCloneBench			OJClone		
Approach	P	R	F1	P	R	F1
Deckard‡	0.93	0.02	0.03	0.99	0.05	0.10
DLC‡	0.95	0.01	0.01	0.71	0.00	0.00
SourcererCC‡	0.88	0.02	0.03	0.07	0.74	0.14
CDLH‡	0.92	0.74	0.82	0.47	0.73	0.57
TBCCD+token(median)	0.97	0.96	**0.97**	0.99	0.99	**0.99**
TBCCD+token(average)	0.97	0.96	**0.96**	0.99	0.99	**0.99**

syntactical information, and it is best to capture syntactical information of source code explicitly rather than implicitly.

Our results indicate that tree-based CNN is more effective than tree-based LSTM in capturing code semantics. One reason for our results may be that there are a lot of layers of nodes in an AST (being a typical case), and the large depth makes even LSTM (designed to improve RNN regarding long-range dependency) unable to remember information that is a long distance away, whereas our approach does not suffer from this issue. We plan to conduct additional experiments to further understand this issue in our future work.

In addition to comparing the effectiveness of our approach with CDLH, we also compare the efficiency. Due to their recurrent structure, RNNs cannot be computed in parallel. As a result, training RNNs takes quite some time, limiting its applications. Compared to RNNs, CNNs generally take much less time to train, since the convolution steps can be carried out in parallel. LSTM is one type of RNN, so our approach using tree-based CNN is likely more computationally efficient than CDLH, which uses tree-based LSTM.

Wei and Li [9] do not report the training time of CDLH in their paper. Due to insufficient details in their paper, we have not been able to completely replicate CDLH. Nevertheless, using our partially replicated version of CDLH, we are not able to finish training on the datasets in two days, whereas we are able to finish training for our approach in several hours.

D. RQ4: Effectiveness of our approach when using training and test sets with different semantic meanings

To test the generalization ability of our approach and to measure the effectiveness of our novel PACE embedding technique, we use the OJClone dataset as described in Section V-B3, and compare the effectiveness of TBCCD+token and TBCCD+token+PACE.

As shown in Table VII, TBCCD+token is the worst performer in the generalization test, in contrast to the finding that it is one of the best performers in RQ1 (Section VI-A). This result is what we expect since TBCCD+token exploits token information with language models and thus likely does not generalize well to code fragments that have substantially different semantic meanings as the code fragments from the training set. Aside from the code fragments with different semantic meanings, we also find that the code fragments in the test sets contain a lot of out-of-vocabulary tokens: there are 3,820 tokens in the training set vocabulary, while there are 9,251 tokens in total that are out-of-vocabulary in the 6 test sets. This poor performance is precisely why we propose PACE. TBCCD+token+PACE consistently outperforms TBCCD+token across the 6 test sets, demonstrating the effectiveness of PACE.

TABLE VII
USING TBCCD+TOKEN AND TBCCD+TOKEN+PACE TO DETECT CLONES IN UNSEEN DATA FROM OJCLONE.

	TBCCD+token			TBCCD+token+PACE		
Question IDs	P	R	F1	P	R	F1
16 ∼ 30	0.17	0.42	0.25	0.35	0.46	0.40
31 ∼ 45	0.26	0.58	0.36	0.51	0.62	0.56
46 ∼ 60	0.21	0.54	0.31	0.59	0.48	0.53
61 ∼ 75	0.27	0.52	0.36	0.58	0.49	0.53
76 ∼ 90	0.25	0.50	0.34	0.61	0.44	0.51
91 ∼ 104 + 16	0.23	0.49	0.31	0.45	0.41	0.43

According to the way that we construct a test set, only $\frac{50\times49\times15}{750\times749} = 6.54\%$ are true clone pairs, indicating that guessing all pairs to be true clone pairs should yield a F1-score of 0.12. Another baseline is 0.14 of SourcererCC, the best F1-score that a traditional approach can get on OJClone, as shown in Table VI. Compared to these baselines, the two variants of our approach both have learned semantic patterns to various degrees and are much better than traditional approaches. Yet, we cannot overlook the fact that even the best performer (TBCCD+token+PACE) drops about 50% in F1-score compared to its stellar performance in RQ1 (from 0.99 to around 0.50). Our finding indicates that the first 15 programming questions in OJClone are quite different than the others, and the 15 programming questions simply may not be enough for neural networks to learn enough patterns in source code. In future work, we plan to conduct more experiments to further investigate this issue and more research efforts to develop a more generalizable and practical approach.

VII. RELATED WORK

A. Traditional Approaches

Most traditional approaches of detecting code clones are targeted at Type-3 clones.

Deckard [14] is a popular AST-based approach that compares two code fragments over a vector representation rather than subtree. Similar to their approach, our approach also converts ASTs into vectors. However, unlike Deckard, which generates vectors using pre-defined rules (each dimension represents the occurrences of a specific tree pattern in the subtree), our vector representations of code fragments are learned in a supervised fashion.

SourcererCC [7] is a token-based approach for detecting code clones. Similar to our approach, SourcererCC detects code clones at the lexical level by comparing subsequences but it does not use the structural information of the code.

Various researchers have proposed other approaches besides AST-based or token-based approaches to detect Type-4 clones. Komondoor and Horwitz [29] use Program Dependency Graphs (PDGs) to detect clones. They represent a program as a dependency graph, and transform the problem of clone detection into finding isomorphic subgraphs over PDGs. PDGs contain a program's control-flow and data-flow information, which carries more semantic information than an AST. However, identifying isomorphic subgraphs is an NP-hard problem, and such PDG-based approach is difficult to scale. Other approaches to detecting semantic clones include using static analysis to extract the memory states for each procedure exit point [30] and using random tests to compare the outputs of two code fragments [31]. Both of these approaches need to compile and run the code, while our approach can be applied statically to uncompilable code.

B. Deep Learning Approaches

With advances in deep learning, the use of deep learning to detect code clones has emerged in recent years.

White et al. [11] propose to first use RNN to learn "term embeddings" of source code and then train a recursive autoencoder to learn the vector representation of an entire AST (after transforming an AST to a full binary tree), thus combing both lexical and syntactical information for detecting code clones. However, as the authors point out in their paper [11], their work is still preliminary and is largely meant to demonstrate the feasibility of using deep learning to detect code clones rather than showing its practicality.

CCLearner [18] is another approach of clone detection based on deep learning. It categorizes tokens into eight categories. For a pair of code fragments (methods), it calculates eight similarity scores in terms of token frequency in each category to form a feature vector, which is then fed to a deep neural network. The feature vector is manually calculated, and the neural network can be replaced by other classifiers. CCLearner is mainly a token-based approach of clone detection based on deep learning.

In order to effectively detect Type-4 clones, Wei and Li [9] formulate code-clone detection as a supervised learning-to-hash problem and propose an end-to-end deep learning approach named CDLH. They use an AST-based LSTM to structurally encode lexical information of a code fragment into a vector and add a hash layer to learn binary embedding for better efficiency when comparing two vectors. In their follow-up work [32], they propose to use unlabeled data to further improve effectiveness and use adversarial training to make the learned model more robust. CDLH well surpasses previous related approaches in detecting Type-3 and Type-4 clones in terms of their F1-scores. As confirmed by our experiments, tree-base convolution can capture code structural information in ASTs more effectively than tree-based LSTM. In fact, our work achieves even better results than CDLH on the two datasets based on what Wei and Li [9] report in their paper.

C. Unlimited-token vocabulary

Unlimited vocabulary size of tokens in source code poses a major challenge to modeling source code [17]. A common technique that can mitigate the challenge is to separate a token into a sequence of tokens according to camel case [33]. Hu et al. [21] propose a new technique to represent the out-of-vocabulary tokens for source code: this technique essentially uses the corresponding AST node type embedding to represent the out-of-vocabulary token instead of the "⟨UNK⟩" token commonly used in NLP. They also use a pair of AST node type and token to represent in-vocabulary tokens. One can find some resemblance in the way that we enhance ASTs with tokens. However, we also propose PACE, an embedding technique that solves the problem of out-of-vocabulary tokens, and have shown its effectiveness in a generalization test.

VIII. Conclusion

Numerous approaches for detecting code clones have been proposed over the last two decades, but few of them are able to effectively detect semantic clones (i.e., clones that are very different syntactically). A recent advance is CDLH [9], which uses AST-based LSTM to learn supervised semantic features to outperform existing related approaches in detecting semantic clones. To fully unleash the power of deep learning for detecting code clones, in this paper, we have presented a novel approach of applying tree-based convolution over a token-enhanced AST to detect semantic clones. Our approach is able to directly capture subtree features to make full use of the structural information in an AST. We have also enhanced an AST by appending tokens to corresponding AST nodes as a way of incorporating lexical information in source code tokens. We have evaluated our approach on two public datasets (BigCloneBench and OJClone), and our experimental results show that our approach can outperform CDLH by a large margin, while being more computationally efficient. To further improve the generalization ability of our approach, we propose a novel technique of token embedding named position-aware character embedding (PACE), which can help alleviate the problem of unseen data in clone detection. PACE enables our approach to be substantially more effective than existing related approaches when detecting code clones in unseen data.

Acknowledgment

This research is supported by the National Key R&D Program under Grant No.2018YFB1003904, and the National Natural Science Foundation of China under Grant No.61832009 and No.61529201, and in part by National Science Foundation under grants no. CNS-1513939, CNS-1564274, CCF-1816615.

References

[1] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *WCRE*, 2006, pp. 253–262.

[2] C. K. Roy and J. R. Cordy, “A mutation/injection-based automatic framework for evaluating code clone detection tools,” in *ICSTW*, 2009, pp. 157–166.
[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” *IEEE Trans. Software Eng.*, vol. 33, no. 9, pp. 577–591, 2007.
[4] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” Queens University, Tech. Rep., 2007.
[5] C. K. Roy and J. R. Cordy, “Benchmarks for software clone detection: A ten-year retrospective,” in *SANER*, 2018, pp. 26–37.
[6] J. Svajlenko and C. K. Roy, “Evaluating clone detection tools with BigCloneBench,” in *ICSME*, 2015, pp. 131–140.
[7] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “SourcererCC: Scaling code clone detection to big-code,” in *ICSE*, 2016, pp. 1157–1168.
[8] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, “Towards a big data curated benchmark of inter-project code clones,” in *ICSME*, 2014, pp. 476–480.
[9] H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in *IJCAI*, 2017, pp. 3034–3040.
[10] K. S. Tai, R. Socher, and C. D. Manning, “Improved semantic representations from tree-structured long short-term memory networks,” in *ACL*, 2015, pp. 1556–1566.
[11] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *ASE*, 2016, pp. 87–98.
[12] “Neural network embeddings explained,” https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526.
[13] “Word2Vec embeddings,” https://radimrehurek.com/gensim/models/word2vec.html.
[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: Scalable and accurate tree-based detection of code clones,” in *ICSE*, 2007, pp. 96–105.
[15] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in *AAAI*, 2016, pp. 1287–1293.
[16] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, “Building program vector representations for deep learning,” in *KSEM*, 2015, pp. 547–553.
[17] V. J. Hellendoorn and P. T. Devanbu, “Are deep neural networks the best choice for modeling source code?” in *ESEC/FSE*, 2017, pp. 763–773.
[18] L. Li, H. Feng, W. Zhuang, N. Meng, and B. G. Ryder, “CCLearner: A deep learning-based clone detection approach,” in *ICSME*, 2017, pp. 249–260.
[19] “OpenJudge,” http://poj.openjudge.cn/.
[20] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, “Summarizing source code with transferred API knowledge,” in *IJCAI*, 2018, pp. 2269–2275.
[21] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation,” in *ICPC*, 2018, pp. 200–210.
[22] S. Jiang, A. Armaly, and C. McMillan, “Automatically generating commit messages from diffs using neural machine translation,” in *ASE*, 2017, pp. 135–146.
[23] “Our dataset and source code,” http://github.com/yh1105/datasetforTBCCD.
[24] “TensorFlow,” http://www.tensorflow.org/.
[25] Z. Yu and G. Liu, “Sliced recurrent neural networks,” in *COLING*, 2018, pp. 2953–2964.
[26] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, “A convolutional neural network for modelling sentences,” in *ACL*, 2014, pp. 655–665.
[27] Y. Kim, “Convolutional neural networks for sentence classification,” in *EMNLP*, 2014, pp. 1746–1751.
[28] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, “Convolutional sequence to sequence learning,” in *ICML*, 2017, pp. 1243–1252.
[29] R. Komondoor and S. Horwitz, “Using slicing to identify duplication in source code,” in *SAS*, 2001, pp. 40–56.
[30] H. Kim, Y. Jung, S. Kim, and K. Yi, “MeCC: Memory comparison-based clone detector,” in *ICSE*, 2011, pp. 301–310.
[31] L. Jiang and Z. Su, “Automatic mining of functionally equivalent code fragments via random testing,” in *ISSTA*, 2009, pp. 81–92.
[32] H. Wei and M. Li, “Positive and unlabeled learning for detecting software functional clones with adversarial training,” in *IJCAI*, 2018, pp. 2840–2846.
[33] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in *ICSE*, 2018, pp. 933–944.