Fast Code Clone Detection Based on Weighted Recursive Autoencoders

JIE ZENG1, KERONG BEN1, XIAOWEI LI2, AND XIAN ZHANG1

1College of Electronic Engineering, Navy University of Engineering, Wuhu, 430033 China
2Jinghang Research Institute of Computing and Communication, Beijing, 100074 China

Corresponding author: JIE ZENG (e-mail: zyj183247166@qq.com).

ABSTRACT Code cloning considerably facilitates software development but also leads to recurring bugs and other software quality problems. In this paper, we propose a fast code clone detection method based on weighted recursive autoencoders (RAE) to measure code similarity at the function level. Different from manually defining features for code clone detection, our deep learning-based method can automatically learn program features. First, we analyze program abstract syntax trees using weighted RAE, extract the program features and encode the functions to vectors. During the modeling process, we consider node weight information in abstract syntax trees to increase the proportion of information contributed by important nodes in the final vector representation of one program. Second, we report functions with similar vectors as code clone pairs. The second phase is time consuming when analyzing large software systems because it needs quadratic pairwise comparisons. To solve this problem, we transform the clone detection problem into an approximate nearest neighbors search (ANNS) in a high-dimensional vector set and use the navigating spreading-out graph to reduce the computational time complexity. Experimental results on BigCloneBench show that our method outperforms the compared algorithm based on unweighted RAE in terms of precision, recall, and AUC value and can return clone pairs in approximately 33 min, while the compared algorithm requires approximately 14 days when performing pairwise comparisons among 785,438 functions' vectors. Our method also outperforms many prominent tools, including Oreo, in detecting Moderately Type-3 or Type-4 clones, and our false positive rate (FPR) equals 0.055, which means few false positives. More importantly, our method has no need for labeled data, and all of the source code is released to guarantee experimental reproducibility.

INDEX TERMS Code Clone, Fast Code Clone Detection, Abstract Syntax Tree, Weighted Recursive Autoencoders, Navigating Spreading-out Graph

I. INTRODUCTION

Code clone detection [1] finds clone pairs that result from copy and paste. Copying and pasting a piece of code with or without modifying often causes two or more pieces of code to resemble each other. This kind of behavior facilitates software development considerably but also leads to bug replication [2], [3]. When one bug occurs in the original code, its cloned code will likely have the same bug, leading to bugs spreading throughout software systems, similar to a cold virus. Although the original bug may be repaired, it is often difficult for the repair process to consider all code pieces cloned from the original. Aiming to address such problems, bug detection technologies based on code clone detection have been widely studied [4-6] and achieved good effects. Therefore, code clone detection as a basic analysis technology is of great significance for maintaining software quality.

Bellon et al. [7] divided code clones into four types, Type-1 to Type-4, according to their similarities. Traditional methods perform code clone detection utilizing different levels of source code representations, including text, token, abstract syntax tree (AST), program dependency graph (PDG), and metric, such as Nicad [8], CCFinder [9], Deckard [10] and SourcererCC [11]. These methods are effective at detecting the first three types of code clones but are still insufficient for detecting Type-4 clones, which have very low syntax similarity. Therefore, choosing features to represent programs and measure
program similarity for detecting Type-4 clones is still difficult.

With the success of deep learning [12] in natural language processing, image processing, and some other fields, it begins to be used in the field of program analysis. One advantage of deep learning is that it can automatically learn features from data. Recursive autoencoders (RAE) were first proposed by Socher et al. [13] to measure the similarity of sentences in natural language. In 2016, White et al. [14] used this model for the first time for code clone detection and detected clones that were undetected by traditional methods, such as Deckard [10]. In 2018, White et al. [15] applied RAE to a variety of source code representations for code similarity analysis, proving its effectiveness.

Traditional methods treat program words discretely and in isolation, while RAE does not. RAE first learns distributed word representations [16] based on statistical language models, such as word2vec [17] and GloVe [18], which can analyze the semantic similarity of different words. For example, the program keywords "for" and "while" are different in literals, but their distributed representations are similar. This characteristic is particularly important for detecting complex clones such as Type-4 clones with low syntax similarity. Although RAE does not use semantic information, such as program control flow and data flow, and conducts analyses based on only ASTs, White et al. [15] found that for the code clone detection, using ASTs leads to a better balance of precision and recall than using program dependency graphs. Different levels of source code representations have advantages and disadvantages for clone detection, which require more scientific and systematic evaluations to determine which representation is better.

RAE is effective at program feature extraction and code clone detection; however, there still exists much room for improvement.

First, RAE can be more accurate for code clone detection. After completing program feature extraction using RAE, White et al. [14], [15] used root node vector representation to measure code similarity but ignored the learned representations of other nodes in the tree structure. Socher et al., who originally proposed RAE, did not measure phrase similarity with only root node information as previously in [13] but calculated the similarities between all the nodes of two grammar trees [19]. However, a single program has many more words (hundreds or thousands) than a natural language phrase, and if the same calculations are performed between two programs, this process will consume considerable computing resources and is not practical. Therefore, the method [19] is not suitable for program analysis, and we need new strategies. In addition, the importance of different AST nodes should be distinguished. For example, a node that invokes a special function is more valuable in measuring code similarity than a generic program keyword because the former reflects the functionality, but the latter is common in programs.

Second, RAE can be more scalable to a large-scale system. White et al. [14], [15] conducted experiments on small datasets. When we applied the same method to analyze a large dataset, BigCloneBench [20], we found that the experiment could not be carried out and was estimated to require 14 days on a desktop computer because of quadratic pairwise comparisons in the set of high-dimensional vectors. SourceCodeCC [11] adopts a filtering mechanism to reduce the number of candidate clone pairs, enabling clone detection on a large code repository GitHub and returns considerable valuable clone information [21]. D-CCFinder enhances scalability based on parallelism [22]. Differing from these methods, RAE encodes programs into a set of high-dimensional vectors, and the task of code clone detection is transformed to searching for similar vectors in the set. When the scale of the vector set is large and the vector dimension is high, it is necessary to perform an approximate nearest neighbor search (ANN) [23] to obtain an approximate solution and return results quickly. Deckard [10] used LSH (locality-sensitive hashing) to complete vector clustering in the final stage of clone detection. However, the authors of Oreo [24] claim that Deckard's experiment on BigCloneBench cannot be carried out because Deckard produces more than 400 GB of intermediate data.

In response to the two challenges, we propose a fast code clone detection method based on weighted RAE. The method consists of two stages: feature extraction by weighted RAE, and fast code clone detection based on an NSG (navigating spreading-out graph) [25]. We assess and prove the effectiveness of our method on the standard benchmark, BigCloneBench. We release all the source code and data of our method\(^1\), which makes the experiments conducted in this paper reproducible.

\(^1\) https://github.com/zyj183247166/Recursive_autoencoder_xiaojie
literals, types, and identifiers.

Type-3: Two code fragments are copied with further modification such as changed, added or removed statements in addition to variations in whitespace, layout, comments, literals, types, and identifiers.

Type-4: Two code fragments perform similar functionality without being textually similar.

Svajlenko [20] further divide the Type-3 and Type-4 into three categories based on their syntactical similarity values: Strongly Type-3, similarity in range [0:7; 1:0), Moderately Type-3, [0:5; 0:7), and Weakly Type-3 (Type-4), [0:0; 0:5).

A. CODE CLONE DETECTION METHODS

Traditional methods manually define features for clone detection and extract them from different source code representations. For example, Nicad [8] analyzes at the text level, targeting the Type-3 clone, which is caused by addition, deletion, modification, and other operations after copying. CCFinder [9] works at the lexical level and can effectively detect Type-2 clones caused by the rename operation after copying. Deckard [10] analyzes ASTs and can detect Type-4 clones. However, these methods all rely on expert experience and cannot automatically determine which features are better for code clone detection. Different from traditional methods, deep learning-based methods can automatically learn program features.

White et al. [14], [15] uses RAE to learn vector representations at the AST level and calculated the similarity of two code fragments by using the vectors of the root node in their trees to represent them. If the Euclidean distance of two vectors is under a set threshold, then the corresponding programs are similar. This paper continues their work and improves the accuracy and scalability of their method.

CCLearner [26] works on the lexical level, divides program tokens into eight categories and then represents programs as token-frequency list vectors. Then, for each code pair, CCLearner computes a similarity score between the token-frequency lists for every token category. Thus, an eight-dimensional vector, each dimension corresponding to a similarity score of one token category, is computed to represent a code pair. Finally, CCLearner performs deep learning on these vectors and trains a binary classifier. Compared with RAE, CCLearner lacks the structural information of AST.

CDLH (clone detection with learning to hash) [27] utilizes AST-based LSTM (long short-term memory) to learn code fragment representation, then learns hash functions to encode these representations into binary hash codes, and finally uses the Hamming distance metric to measure the similarity of a code pair. Similar to traditional RAE, CDLH uses the root node representation to represent a code fragment and ignores the learned representation of other nodes in the tree structure.

DeepSim [28] encodes the control flow and data flow of a code fragment into a semantic matrix, based on which a deep neural network is designed to measure code functional similarity. It works well for detecting Type-4 clones. Different from work based on program control and data flow, we perform code clone detection on ASTs, similar to White et al., CDLH, and Deckard. In addition, White et al. [15] found that performing clone detection on ASTs has a better performance balance than program control flow. We believe these two representations are complementary.

Oreo [24] represents a code fragment using 24 kinds of metrics, including the number of declared variables, then trains a binary classifier using a vector of 48 dimensions, which corresponds to a pair of code fragments as an input into a symmetrically structured Siamese network [29]. DeepSim and Oreo maintain the symmetry of their networks by using the pooling layer and sum operation, respectively. The symmetry here denotes that the network will respond the same when directly accepting a pair of code fragments as input or first changing the inner order of the pair before accepting it as input. Symmetry is very important for clone detection, and our method is also symmetric without requiring a pooling layer or other operations because the clone detection model based on RAE uses the Euclidean distance metric which is naturally symmetric and not affected by the inner order of a pair.

Code clone detection methods based on deep learning can be summarized, as shown in Table 1, from the perspectives of analyzed program information, built network structure, and the presence or absence of supervisory signals back-propagated from labeled data. Although CCLearner analyzes ASTs to obtain word frequency sequences, it does not use the tree structure information and remains at the lexical level. Among the detection models based on AST, traditional RAE and CDLH both use the learned feature vector of only the root node to complete the final binary classification and ignore the learned representations of other nodes in the tree structure. This problem is what we take measures to solve in this paper.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Analyzed Program Information</th>
<th>Built Network Structure</th>
<th>Supervised or Unsupervised</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLC [14]</td>
<td>Token and AST</td>
<td>Traditional RAE</td>
<td>Unsupervised</td>
</tr>
<tr>
<td>CCLearner [26]</td>
<td>Token</td>
<td>Deep Feedforward Network</td>
<td>Supervised</td>
</tr>
<tr>
<td>CDLH [27]</td>
<td>AST</td>
<td>AST-based LSTM</td>
<td>Supervised</td>
</tr>
<tr>
<td>DeepSim [28]</td>
<td>Program Control and Data Flow</td>
<td>Symmetrical Deep Feedforward Network</td>
<td>Supervised</td>
</tr>
</tbody>
</table>
Finally, RAE does not need labeled data and is easier to use in application scenarios without labeled datasets.

B. SCALABILITY OF METHODS BASED ON DEEP LEARNING

Many code clone detectors face scalability issues [11]. That means these detectors are not scalable to a large-scale software system that may include hundreds of millions of lines of code. Different from SourcererCC [11] and D-CCFinder [22], which are scalable, methods based on deep learning use high-dimensional vectors that represent corresponding code fragments.

Currently, White et al. [14], [15] have not evaluated RAE on a large dataset. Although the CDLH and DeepSim methods have been evaluated on BigCloneBench, they directly use the labeled clone pairs as inputs to their networks and discard unlabeled functions. BigCloneBench labels approximately only one-tenth of all functions in the bcd\_reduced source repository, which was obtained by reducing the IJaDataset 2.0 [20]. Thus, if CDLH and DeepSim are evaluated directly on bcd\_reduced and not only on labeled code pairs, there will be approximately three hundred billion code pair inputs, which is considerably more than the eight million pairs they analyze.

Oreo and CCLearner use filtering mechanisms based on a hypothesis to reduce the size of candidate clone pairs. For example, they both determine that a code pair cannot be a clone if the two fragments of this pair are very different in code size. Although filtering enhances scalability for large software systems, this filtering sometimes leads to deviations. For example, in the BigCloneBench dataset that Oreo uses, the two functions numbered 8,273,323 and 1,356,3706, which are a Type-4 clone pair. The former function is located in 672942.java starting at line 44 and ending at line 49, while the latter function is located in 625451.java starting at line 44 and ending at line 193. According to the filtering mechanisms based on code size, both Oreo and CCLearner will not consider this pair as a code clone.

Therefore, we need new strategies to eliminate scalability issues when using vector-based methods. Our method is vector-based, similar to the above methods; however, the difference is that to solve the scalability issue, we transform the problem of clone detection into an ANNS [23] in a high-dimensional vector set in the final stage of our method. The traditional Deckard [10] method works on vectors extracted from ASTs and uses LSH to cluster vectors. However, the Oreo [24] authors claim that Deckard cannot be evaluated on BigCloneBench because it produces 400 GB of immediate data, which cannot be disposed of by BigCloneEval [30]. In addition, the efficiency of the LSH it uses has been exceeded by many algorithms such as the NSG [25] algorithm and it uses manually defined features.

Currently, many algorithms and standard evaluation sets for ANNS have been designed [31] that can support large-scale retrieval of high-dimensional vector data such as images, recommendations, and multimedia. From many algorithms, we chose the NSG [25] algorithm as the ANNS solution because it achieves better performance than LSH and some other algorithms.

C. BENCHMARK FOR CODE CLONE DETECTION AND EVALUATION PROCEDURE

Early code clone detection methods often used their own datasets, which lack unity, for evaluation. To address this problem, Svajlenko et al. [20] built the BigCloneBench dataset specifically for evaluating code clone detection methods. They built the dataset through code search and manual annotation based on the repository IJaDataset 2.0, which contains 25,000 Java projects. Currently, BigCloneBench includes a total of 8,584,153 true clone pairs from Type-1 to Type-4 (BigCloneBench names Type-4 as Weakly Type-3) and 279,032 false clone pairs (labeled negative). BigCloneBench works at function granularity and labels clone pairs and has expanded from covering ten functionalities initially to the current 43 functionalities. For ease of evaluation, BigCloneBench built a new source code repository named bcd\_reduced that is smaller than IJaDataset but still contains a large amount of code.

It is impossible for BigCloneBench to manually label all the function pairs in the large-scale repository; thus, many clone pairs reported by the evaluated clone detectors are not labeled, and therefore, the precision factor can only be estimated. Paired with BigCloneBench is the BigCloneEval [30] tool, which measures the recall factor of clone detectors and does not yet support computing other indicator factors. Here, recall is the ratio of the clones tagged by BigCloneBench that a tool can detect, and precision is the ratio of the clones reported by a tool that are true clones, not false positives.

Finally, there are two methods for measuring clone detectors on BigCloneBench:

1) EVALUATION METHOD 1

In this method, clone detectors work directly on the source code repository and report clone pairs and then reported clone pairs are compared with the labeled clone pairs [30]. This method requires that the evaluated methods are scalable to a large-scale code repository and precision can only be estimated because some unlabeled clone pairs may be reported. Both Oreo and CCLearner use this method. However, they do not estimate precision using the method described by Svajlenko et al. [20]; they estimate precision by sampling inspection.

2) EVALUATION METHOD 2

In this method, clone detectors work directly on labeled clone pairs, and the unlabeled functions are discarded, similar to CDLH [27], DeepSim [28]. This method is simple, and clone detectors do not need to consider scalability on the large code repository bcd\_reduced. Therefore, this method is better for evaluating the clone
detector’s classification capacity rather than scalability.

III. THE OVERALL FRAMEWORK OF OUR METHOD

Our goal is to enhance both the accuracy and scalability of traditional RAE for code clone detection.

As shown in Figure 1, the overall framework is divided into two phases, namely, program feature extraction and fast ANNS. Through our framework, we transform the code clone detection problem to an ANNS.

In the first phase, we extract program features at function granularity and transform a program to a high-dimensional vector, which includes three steps: data preparation, data preprocessing, and program feature extraction based on weighted RAE.

In the data preparation step, we need to specify the target software system for performing clone detection and divide the training dataset for deep learning. In the data preprocessing step, we analyze ASTs of all functions in the target system and transform them into full binary trees according to the rules proposed by White et al. [14]. We also add seven rules based on their work. Additionally, sentences composed of words are generated, and one sentence corresponds to one function, with numeric and string constants in functions that have been standardized to a normalized word.

In the program feature extraction step, statistical learning models are first used to generate a word vector for each word in the sentence corpus, and then RAE is trained to reduce the overall reconstruction error on the training corpus until the trained model fits. After that, we weight different nodes in ASTs and generate program vectors based on the weighted RAE.

After the first phase of feature extraction, every function in the target system is transformed into a vector. In the second phase, we dispose of the set of vectors and report clone pairs. We use the Euclidean distance metric to evaluate the similarity between vectors. If the distance between two vectors is less than the set threshold, the two corresponding functions are considered a clone pair; otherwise, they are not. Because this procedure needs quadratic pairwise comparisons in the vector set, when the number of functions in the analyzed software system is large, it will be very expensive in time cost. To solve this problem, we use the NSG algorithm to search for neighboring vectors and return clone results.

IV. THE DATA PREPROCESSING

In the field of program analysis, Mou et al. [32] proposed using the continuous binary tree model to build RAE, which could conduct deep learning directly on multibranch AST. However, RAE used in our method follows the idea of White et al. [14], [15]. That is, the AST of a function is first converted into a full binary tree according to some transformational rules and then input to the RAE. The advantage of this approach is that not all nodes of the AST are reserved except for those nodes that are important for measuring code similarity, which benefits RAE training.

Thus, our weighted RAE needs training data with the structure of a full binary tree. For each function, we first transform its AST to a full binary tree and then generate a sentence consisting of tokens.

A. TRANSFORMING AST INTO A FULL BINARY TREE

White et al. [14] proposed 25 transformation rules for generating a full binary tree from a Java AST. These rules are mainly for processing nonterminal nodes with more than two child nodes and transforming them into nonterminal nodes with 1 or 2 child nodes. This procedure is called Case-II conversion. After this procedure, there still exist some nodes with 1 child node. To process these nodes, the nodes are merged with their only child by retaining the more important node; this procedure is called Case-I conversion. After the two kinds of conversions, an AST is transformed into a full binary tree. CDLH [27] also converts ASTs to full binary trees when using AST-based LSTM but does not give detailed explanations about the transformation rules it uses.

We continue the work of White et al. but identify another 7 node types that cannot be processed, leading to the failure of full binary tree generation. To solve this problem, we add other 7 transformation rules. The Java AST node type

---

1 https://sites.google.com/site/deeplearningclone/binary-grammar.pdf?attredirects=0
2 https://github.com/micheletufano/ast2bin
TypeParameter is taken as an example here, and the case-II conversion rule for processing TypeParameter nodes is shown in Table 2.

During the transformation process, relevant nodes can be extracted from the children of TypeParameter nodes according to the corresponding Case-II conversion rule. TypeParameterHeader, ModifierList, and TypeList are types we manually build and specify to complete the conversion. The example case-II conversion produces a nonterminal node with artificial type TypeList and its only child node with type Type, which originally exists in Java Language Grammar. To process this generated node, a Case-I conversion is performed.

### B. GENERATING SENTENCES FOR FUNCTIONS

In the full binary tree, each leaf corresponds to a program word, and each nonterminal node does not. We use postorder traversal\(^5\) to process the whole tree, arranging the leaf node words into a sentence. This approach standardizes the program and eliminates the interference of coding style, whitespace, newline, and other unrelated factors. We transform string constants, character constants, integers and floating-point numbers to `<string>`, `<char>`, `<int>` and `<float>`, respectively. Table 3 presents a sample Java function named `print`, along with its generated sentence and binary tree (recorded in the text).

The first six-tuple in the text recording binary tree structure of function `print` represents that the second-word void and the third-word print have a common parent node numbered 11 with type SignatureElementList. We use TERMINAL_NODE_TYPE to indicate that one node is a leaf node. Except for the artificial types, the remaining types all exist in the Java Language Grammar which can be analyzed using the Eclipse JDT toolkit\(^6\).

The above generation procedure is based on an improved form of ast2bin offered by White et al [14]. We adopt a new method to record the binary tree for our weighted RAE that is explained in the next Section.

### V. FEATURE EXTRACTION BASED ON WEIGHTED RAE

The main goal of weighted RAE is to increase the proportion of information contributed by more important nodes in the final vector representation of one program to improve the representation accuracy. To achieve this goal, we first weight different nonterminal nodes in one AST. Second, after the model converges, we do not use the root node vector representation to represent the corresponding program as in traditional RAE but perform a weighted average of all nonterminal node vector representations.

### A. VECTOR GENERATION FOR LEAVES

Program tree leaves correspond to specific words. Words are the basic components of a program, and all changes between two programs are ultimately the changes between words. Therefore, the program similarity analysis inevitably

---

\(^{5}\) [https://en.wikipedia.org/wiki/Tree_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

\(^{6}\) [https://www.eclipse.org/jdt/](https://www.eclipse.org/jdt/)
requires word similarity analysis. On the corpus containing all sentences, we use the statistical language model, word2vec [17], to learn the vector of each word. By learning the distributed word vector representations, the model makes the words with different semantics apart from each other in the vector space and the words with similar semantics (such as "for" and "while") closer together in the vector space.

B. WEIGHTING NONTERMINAL NODES USING TF-IDF

We use information in ASTs to complete code clone detection. AST leaves corresponding to words can be modeled to vector representations by statistical language models, but nonterminal nodes cannot be modeled because they do not correspond to specific words and own only a type. To use the information contained in these nonterminal nodes to distinguish between different programs, we distinguish nonterminal nodes by their types.

For example, we processed the corresponding binary tree of the function `print` in Table 3 in a postorder traversal mode, and arranged all nonterminal node types into a text sequence, as shown in Figure 2.

![Figure 2. Text sequence of all nonterminal types in the example function’s AST. The corresponding tree is processed in the postorder traversal.](image)

To distinguish the importance of different nonterminal nodes to the program, we need to weight their types. Here, we refer to the term frequency-inverse document frequency TF-IDF (term frequency-inverse document frequency) model.

The TF-IDF model is one basic model for text similarity measurement. The higher the TF-IDF value of a term, the more important it is to the text. Given a set of texts, the TF-IDF value of one term is computed based on two factors:

- **Term Frequency.** The more times a term appears in one text, the more relevant it is to the text.
- **Inverse Document Frequency.** The more texts a term appears in, the lower its ability to distinguish between different texts is.

After a function is converted to a text sequence of nonterminal types, we can perform a similar analysis of each term in the sequence from the perspectives of term frequency and inverse document frequency. For example, given a Java function with several while loops and a function without while loops, there will exist several nonterminal nodes with type `WhileStatement` in the former function but not in the latter function. Thus, type `WhileStatement` is more relevant to the former function. In addition, if type `WhileStatement` appears in a large number of different functions, we can also assume that it has a low ability to distinguish between different functions.

Therefore, how important a nonterminal type is to the function where it appears can be computed, and the same nonterminal type may be important to one function but not important to another function. We converted all functions in the target software system to text sequences of nonterminal types and calculated the TF-IDF value of each nonterminal type for each function based on the set of sequences.

C. PROGRAM VECTOR GENERATION BASED ON WEIGHTED RAE

RAE [13] were first proposed for measure the similarity of sentences in natural language and White et al. [14] used this model for the first time for code clone detection. For a program and its full binary tree, which is transformed from its AST, RAE continuously merges and reconstructs vectors of child nodes at each nonterminal node recursively and bottom-up starting from leaves and ending at the root node. The goal is to minimize reconstruction errors that occur at all nonterminal nodes, including the root node and finally calculate their semantic vector representations. The learned representations combine the structural information of the AST and can effectively represent the program.

Differing from traditional RAE, the weighted RAE we propose incorporates the weight information of each nonterminal node to increase the proportion of information contributed by more important nodes in the final vector representations. The weighted RAE here is different from that in [13], which weights different reconstruction errors and amplifies the reconstruction errors of more important nodes during the training process. In our experiments, we find that the weighted model in [13] is not appropriate for code clone detection, and its performance is even worse than that of the unweighted RAE. Unlike the weighted RAE from [13], the main idea of our proposed weighted RAE is that we first use the trained RAE to encode the full binary tree of a program and obtain a vector representation for each node in the tree; then, we generate the final representation for the whole tree by considering all nonterminal nodes and their weights rather than using the root node only. The weights of different nodes in ASTs are calculated based on the TF-IDF model.

Each unit of the RAE is an autoencoder, including encoding and decoding phases. An autoencoder is one type of technology for data dimension reduction and can extract the most expressive factors from high-dimensional data. A classic autoencoder is shown in Figure 3, which is composed of an input layer, a hidden layer, and an output layer. It is trained in an unsupervised learning manner, using the gradient descent and backpropagation algorithms to minimize the reconstruction error calculated by comparing the input and output of it.
Given an input \( x \in \mathbb{R}^n \), the output of the hidden layer is represented as follows:

\[
h(x) = f(W^{(1)} \cdot x + b^{(1)}) \quad (1)
\]

\( h(x) \in \mathbb{R}^m \), \( m < n \), and the output of the output layer is represented as follows:

\[
y = W^{(2)} \cdot h(x) + b^{(2)} \quad (2)
\]

\( y \in \mathbb{R}^\ell \) and the dimension of it is the same as the dimension of \( x \). When the model converges, \( h(x) \) is the compressed representation of \( x \), where \( W^{(1)} \in \mathbb{R}^{m \times n} \) and \( W^{(1)} \in \mathbb{R}^{m \times m} \) are the weight matrices and \( b^{(1)} \in \mathbb{R}^m \) and \( b^{(2)} \in \mathbb{R}^\ell \) are bias matrices. These parameters are the ones that need to be learned, and all autoencoders share their same values. \( f \) is a nonlinear activation function, which generally uses the sigmoid or tanh function. The reconstruction error is as follows:

\[
E(\theta) = \| x - y \|^2 \quad (3)
\]

\( \theta = \{W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}\} \).

Then, for the program corpus \( C \), given a program \( P \in C \), assuming that it has \( p \) words (corresponding to the leaves of its tree), the RAE performs encoding and decoding operations at each nonterminal node, as shown in Figure 4. Leaf nodes are represented as their corresponding word vectors, and nonterminal nodes are represented as semantic vectors by encoding their two child nodes’ vectors.

Take the nonterminal node \( O^{(1)} \) as an example; the corresponding vectors of its two child nodes \( \text{leaf}_1 \) and \( \text{leaf}_2 \) are \( x_1 \in \mathbb{R}^d \) and \( x_2 \in \mathbb{R}^d \), then the autoencoder input at \( O^{(1)} \) will be \( x^{(1)} = [x_1; x_2] \in \mathbb{R}^{2d} \). Using Formulas (1), (2), and (3) above, we can compute the hidden layer output \( h^{(1)} \), output \( y^{(1)} \) of the output layer, and the reconstruction error \( E^{(1)}_\theta \). \( h^{(1)} \) is normalized by length [13] and used as the semantic vector representation of \( O^{(1)} \).

With \( O^{(1)} \) and \( \text{leaf}_1 \) as the child nodes of \( O^{(2)} \), the autoencoder input at \( O^{(2)} \) is \( x^{(2)} = [h^{(1)}; x_1] \in \mathbb{R}^{3d} \), where \( x_i \in \mathbb{R}^d \) is the word vector of \( \text{leaf}_i \). Then, the semantic
vector of $O^{(2)}$ is computed, and similar computations repeat until the vector of root node $O^{(r-1)}$ is computed.

Finally, for the corpus $C$, our goal is to optimize the final loss $\Phi(\theta)$ as follows:

$$\Phi(\theta) = \sum_{P \in C} \sum_{i \in \{1, 2, \ldots, p-1\}} \epsilon_{\theta}^{(i)} + \frac{\lambda}{2} \left( \|W^{(1)}\|_2 + \|W^{(2)}\|_2 \right)$$  \tag{4}

$\theta = \{W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}\}$ are the model parameters we need to optimize. $P$ represents that the program $P \in C$ has $p$ program words, and $\epsilon_{\theta}^{(i)}$ is the autoencoder reconstruction error at the $i$-th nonterminal node in the tree structure of $P$. The latter term in Formula (4) is L2 norms for neural network weights, to avoid the model overfitting, where $\lambda$ is a hyperparameter.

After the model converges, we encode the tree structure of the program $P$ from bottom to top and calculate the semantic vectors $\{\text{Vector}^{(1)}, \text{Vector}^{(2)}, \ldots, \text{Vector}^{(r-1)}\}$ for all of its nonterminal nodes.

Assume that the weights of all nonterminal nodes in the program $P$ are $\{\text{Weight}^{(1)}, \text{Weight}^{(2)}, \ldots, \text{Weight}^{(r-1)}\}$ (see Section V-B). Then, the coefficient to be multiplied by the vector of each nonterminal node is

$$f^{(i)} = \frac{\text{Weight}^{(i)}}{\sum_{j \in \{1, 2, \ldots, p-1\}} \text{Weight}^{(j)}}$$  \tag{5}

The sum of all coefficients is 1. Using Formula (5), we obtain each nonterminal node’s importance. Then, the final vector representation for the whole tree is

$$\text{Vector}(P) = \sum_{i \in \{1, 2, \ldots, p-1\}} f^{(i)} \text{Vector}^{(i)}$$  \tag{6}

Here, we use the coefficients because we believe that different nonterminal nodes have different importance, and the computation of the final representation for the program should take this into account.

Unlike our method, traditional RAE uses only the vector of the root node as the final representation of the whole program, ignoring the representations learned for other nonterminal nodes.

VI. FAST CODE CLONE DETECTION BASED ON NSG

For the target software system, every function is encoded to a vector representation by the weighted RAE, forming a set of high-dimensional vectors. Then, we can directly compare two functions by measuring the Euclidean distance of their corresponding vectors and determine that they are a true clone pair if the distance is less than or equal to the set threshold or they are a false pair if the distance is larger than the threshold. Thus, the clone pairs in the target software system can be detected at function granularity.

Assume the target system owns $n$ functions, the time complexity of quadratic pairwise comparisons in the vector set is $O(n^2)$. When $n$ is too large, the computational process will take considerable time because vectors are distributed in the space with hundreds of dimensions and it will take hundreds of floating-point operations to calculate the Euclidean distance of each pair of vectors. To solve this problem, we use the NSG algorithm [25] to complete the ANNS, which means that we do not compare all pairs of a vector set but can also give an approximate solution for searching for neighboring vectors.

The NSG algorithm has superior performance among similar algorithms. An earlier solution to complete the ANNS is to search in the KNN (k-nearest neighbor) graph [33]. For a query vector, the search starts from a random node in the KNN graph and checks its neighbors to find a node closer to the query vector. Then, the found nodes are new starting points for the search, and thus, the average out-degree of the KNN graph determines the retrieval performance. The NSG algorithm reduces the average output-degree by eliminating the redundant edges in the graph using navigating spreading-out, and thus, considerably reducing the search time.

However, the NSG algorithm can return only the nearest neighbor vectors for the query vector. We need to improve the NSG algorithm for the task of code clone detection. Algorithm 1 below describes our procedure of fast code clone detection based on NSG. We first build a NSG graph based on the vector set, and then search neighboring vectors for each vector in the graph, and at last screen out the code clones from the candidates using the Euclidean distance metric and the set distance threshold.

Algorithm 1 NSG-based Code Clone Detection

**Input:** $V$: vectors, $r$: threshold

**Output:** CP: ClonePairs

**Procedure:**

1. Begin
2. build NSG $g$ from $V$
3. $CP = \Phi$
4. repeat pick a $v$ from $V$
5. $results = queryNSG(v, g)$
6. repeat pick a $r$ from results
7. if distance($v, r$) $\leq t$
8. $CP \leftrightarrow CP \cup (v, r)$
9. end if
10. until result $= \Phi$
11. Until $V = \Phi$
12. return $CP$
13. End

VII. EXPERIMENTAL EVALUATION

A. DATASET AND EVALUATION INDICATORS

We evaluate our method on BigCloneBench, a standard benchmark for code clone detection. The dataset consists of two parts: the source repository bcb_reduced and the tag database. bcb_reduced has 43 subfolders, each of which corresponds to a functionality, and stores 55,499 Java files. BigCloneBench uses the tuple (type, name) as a Java file unique identifier; the function is identified by Function_ID, and function location information is recorded by the quadruple (type, name, startline, endline). type represents the
source of the Java files [20], containing three values: ‘sample’, ‘selected’, and ‘default’. startline and endline represent the function’s location. The tag database marks 8,584,153 true clone pairs and 279,032 false clone pairs, with each pair corresponding to a pair of functions. As the same Java file may contain multiple tagged functions that implement different functionalities, some Java files with the same (type, name) are copied to multiple different subfolders in bcb_reduced. Of the 55,499 Java files, there are 50,532 Java files with a unique (type, name).

Because precision can only be estimated on BigCloneBench (see Section II-C), the creators of the dataset mainly measure recall when evaluating ten code clone detectors [34]. Some researchers [24], [26] estimate precision via random sampling, and some researchers [27], [35] directly use the tagged clone pairs. In the latter method, the samples with positive and negative labels in BigCloneBench are extremely unbalanced and if a clone detector tags an arbitrary function pair as a true clone pair, its precision on BigCloneBench can still reach 96.85%. Therefore, we further plot the receiver operator characteristic (ROC) curves to evaluate our method. The ROC curve consists of two parts, true positive rate (TPR) and false positive rate (FPR). The former is the same as recall in this scene, and the latter is calculated by Formula (9). The two indicators measure, respectively, the ratio of positive alarms in the analysis of samples with the positive label and the ratio of false alarms in the analysis of samples with the negative label. Therefore, the class imbalance of BigCloneBench does not affect the ROC curve.

\[
\text{Precision} = \frac{TP}{TP + FP} \tag{7}
\]

\[
\text{TPR} = \text{Recall} = \frac{TP}{TP + FN} \tag{8}
\]

\[
\text{FPR} = \frac{FP}{FP + TN} \tag{9}
\]

TP represents the number of pairs that are tagged as true clone pairs and detected as true, FP represents the number of pairs that are tagged as false clone pairs but detected as true, TN represents the number of pairs that are tagged as false clone pairs and detected as false, and FN represents the number of pairs that are tagged as true clone pairs but detected as false.

B. EXPERIMENTAL DESIGN
To evaluate our method, we design two experiments to answer the following two research questions:

RQ1. Can our method based on the weighted RAE be more accurate for code clone detection than that based on the unweighted RAE [14], [15]?

RQ2. Can our method be scalable to a large-scale software system and return clone detection results in a relatively short time?

The two experiments are as below:

EXPERIMENT1. Referring to the second evaluation method in Section II-C, we evaluate the unweighted and weighted RAE directly on the tagged clone pairs in BigCloneBench.

EXPERIMENT2. Referring to the first evaluation method in Section II-C, we apply our method to the source repository bcd_reduced to evaluate the scalability of our method for large-scale software systems.

C. HARDWARE AND SOFTWARE CONFIGURATION
The experimental hardware configuration mainly includes the following:

- Intel(R) Core(TM) i7-8700K CPU
- NVIDIA GeForce GTX 1080 Ti (discrete graphics)
- RAM 32.00 GB

The main software configuration of the experiments is explained in Table 4. As the source code of NSG can be compiled and used only in Linux, so we use VMware to install a virtual machine with Ubuntu as the operating system.

D. STEPS IN EXPERIMENT1
1) STEP 1: PREPROCESSING PROGRAM DATA
We extract 785,438 functions with more than 1 line and more than 2 words from the 50,532 unique Java files of bcb_reduced. On the basis of 25 case-II conversion rules in ast2bin proposed by White et al. [14], we add another 7 rules. We then transform all AST functions into full binary trees, extract the sentence representation and the corresponding binary tree structure of each function, and record the locations of all functions (see Section IV).

We store all the sentences in the file corpus.txt, all the binary tree structures in the file ast_construction.txt, and all the locations in the file writer_path.txt.

2) STEP 2: PROCESSING BIGCLONEBENCH DATABASE
The source repository bcb_reduced has a total of 785,438 functions, of which 73,319 are tagged by the BigCloneBench.

We find that some of the tagged functions are marked with different locations in BigCloneBench and our location file writer_path.txt. For example, the upload function of the Java file 88026.java is located at lines 52 to 69 in the BigCloneBench, while located at lines 47 to 69 in our writer_path.txt because we use Eclipse's JDT tool to extract Java functions based on AST analysis, which includes comment lines with the sign @. However, these comment lines are not included by BigCloneBench.

Svajlenko et al. [30] used BigCloneEval to determine whether a function detected by the evaluated clone detector is the function tagged by BigCloneBench in the following ways:

Suppose BigCloneBench records a function’s location with the quadruple \((\text{type}=t_b, \text{name}=n_b, \text{startline}=s_b, \text{endline}=e_b)\) (see Section VII-A) and the evaluated detector records a function’s location with the quadruple \((\text{type}=t_w, \text{name}=n_w, \text{startline}=s_w, \text{endline}=e_w)\). Then, only if \(t_b\) is the same as \(t_w\), \(n_b\) is the same as \(n_w\), and Formula (10) is true, the two functions are assumed to be the same function.
Among which 987,474 are unique words. Vectors of 987,464 corpus.txt are repetitively tagged by BigCloneBench and explain in the website7. We also remove the clone pairs, which are repetitively tagged in BigCloneBench and explain in the website7.

Table 4. The main software configuration for our experiments.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Software Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 10</td>
<td>ast2bin, Scikit-learn, VMware Workstation 15 Pro</td>
<td>Python's integration platform, including tools such as Spyder, and various packages such as NumPy.</td>
</tr>
<tr>
<td>Ubuntu 18.04</td>
<td>Efanna9, NSG10, Yael11</td>
<td></td>
</tr>
</tbody>
</table>

\[
\frac{\min(e_w, e_b) - \max(s_w, s_b)}{e_b - s_b + 1} \geq 0.7 \tag{10}
\]

We decide whether the function extracted by our method is the function tagged by BigCloneBench in the same way. Finally, there are 24,816 functions whose positions were slightly different from those marked in BigCloneBench, and 48,478 functions the same. Another 25 functions are wrongly marked by BigCloneBench, which we judge manually and explain in the website7. To avoid the evaluation be impacted by these wrongly marked functions, we remove these functions and those clone pairs involved with them from the tables CLONES and FALSE_POSITIVES in the tag database of BigCloneBench. We also remove the clone pairs, which are repetitively tagged by BigCloneBench and explain in the website7.

After the above process, the number of true clone pairs tagged in CLONES is reduced from the original 8,584,153 to 8,575,774, with 3,740 involved with incorrect functions and 4,639 are repetitively tagged. In addition, the number of false clone pairs tagged by FALSE_POSITIVES is reduced from the original 279,032 to 278,961, with 71 pairs involved with incorrectly marked functions.

Finally, we use the remaining 8,854,735 tagged clone pairs in BigCloneBench as the final tagged database to evaluate our method.

3) STEP 3: TRAINING WORD VECTORS

We use word2vec [17] to train word vectors on the file corpus.txt. The file contains a total of 31,937,932 words, among which 987,474 are unique words. Vectors of 987,464 words are learned by word2vec, while another 10 words are automatically filtered out by word2vec because they are too long, such as the word that is 142 characters long beginning with _005 at line 56 of the 102108.java file. For these ten words, we unify their word vectors to the zero vector. The purpose is to ignore their contributions to the final program vector representations and maintain the integrity of the binary tree structure in which the words are located, and thus the weighted RAE can still be computed on these trees.

4) STEP 4: TRAINING RAE

With 25 incorrectly marked functions removed, there remains 73,294 functions marked by BigCloneBench. To avoid any significant favorable bias in our experiments, we do not use these functions during training.

We randomly select 5,000 functions that are not marked by BigCloneBench and extract the corresponding sentences from the file corpus.txt and binary tree structures from the file ast_construction.txt. We remove those functions with more than 300 words and use the remaining 4,770 functions as training data. The purpose is that when the sentence is too long, the binary tree structure will be too large, which will lead to an OOM (out of memory) exception reported by TensorFlow because the GPU (graphics processing unit) memory is exhausted. We randomly select another 5,000 functions as the validation data. Here, there is no upper limit on the sentence length because the calculation process does not require complex operations such as derivations. For each training sample and validation sample, we extracted the vectors of all words and the binary tree structure and input them into the RAE network structure for training. When the model reconstruction error on the validation set cannot be further reduced, the model converges, and the training process is completed.

5) STEP 5: EVALUATION ON TAGGED CLONE PAIRS

For each clone pair in the tagged database, we use the trained model to compute the vector representations of two functions.

---

1 https://github.com/zyj183247166/Recursive_autoencoder_xiaojie/blob/master/wrongLocationMarkedByBigCloneBench.txt
2 https://github.com/zyj183247166/Recursive_autoencoder_xiaojie/blob/master/duplicate_truc_pair_record.txt
3 https://github.com/ZJULearning/efanna_graph
4 https://github.com/zyj183247166/Recursive_autoencoder_xiaojie/blob/master/wrongLocationMarkedByBigCloneBench.txt
5 http://yael.gforge.inria.fr/gettingstarted.html

---

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.
If the Euclidean distance of their vectors is less than or equal to the set threshold, it is reported as a true clone pair; otherwise, it is not. Then, we calculate the precision and recall (see Section VII-A). We compute the recall on tagged clone pairs of each type (from Type-1 to Type-4).

White et al. [14] empirically set thresholds for different software systems. To observe changes in performance along with changes in the set threshold, we plot the P-R curves and ROC curves.

E. STEPS IN EXPERIMENT2

Steps from 1 to 4 in EXPERIMENT2 are the same as those in EXPERIMENT1.

In Step 5 of EXPERIMENT2, we need to evaluate the scalability of our method to large-scale software systems. The source repository bcd_reduced of BigCloneBench has approximately 308.4 billion function pairs, whereas BigCloneBench tags only 8,854,735 function pairs. In EXPERIMENT1, our method is directly evaluated on tagged clone pairs. Differing from it, in EXPERIMENT2, our method is first evaluated on bcd_reduced, and then the detected clone pairs are compared with the tagged clone pairs in BigCloneBench to compute recall, estimate precision, and so on.

As to the 785,438 functions in bcd_reduced, we calculate 785,438 vectors using the trained model with each vector for each function, forming a high-dimensional vector set. Then, according to Algorithm 1 in Section VI, we perform code clone detection based on NSG. Finally, we calculate the performance indicators referring to the first evaluation methods in Section II-C and record the time cost.

F. HYPERPARAMETER SET-UP

Hyperparameters used in RAE are first set empirically, then tuned in experiments, and finally fixed. The maximum number of training epochs is 30. If the reconstruction error on the validation set does not decrease in a consecutive five epochs, the model is considered to be convergent. The stochastic gradient descent algorithm is used to optimize the model, the initial learning rate is set to 0.01, and if the reconstruction error on the validation set does not decrease, the learning rate is reduced by two-thirds. Finally, the model with the lowest reconstruction error on the validation set is saved for evaluation.

In RAE, the dimension of the vector representing a program is the same as that of the word vector. Under the restriction of the physical memory, the higher the dimension we set when using word2vec, the more accurate the vector learned for each word is. However, in the final phase of our method, we need to use an open source project that implements NSG [25] to retrieve similar vectors, and the project uses AVX (advanced vector extensions) instructions to speed up, which requires that the dimension of floating-point vectors must be multiples of 8. Thus, we set the dimension in word2vec as 296, and there are 296 nodes in the hidden layer in each autoencoder of RAE.

The batch size is set differently between the training and validation processes. The training process involves backpropagation and derivative calculation, which have a large computational overhead. If the batch size is too large during training, an OOM exception will occur in the GPU, so we set the batch size to 10. In the validation process, there is no need to perform complex operations, so the load cost is small, and we set batch size to 400 to reduce the overall calculation time.

G. RESULTS AND DISCUSSION

1) RECONSTRUCTION ERROR

Both the unweighted RAE and our weighted RAE use the same neural network model, but the difference is that our method considers vector representations of all nonterminal nodes in ASTs and their weights after the RAE model is trained. The RAE model is trained for a total of 30 epochs, with a total time of approximately 10 hours. In the last epoch, the reconstruction error on the validation set reaches the optimum. Figure 5 shows the changing reconstruction errors on the validation set as the training process proceeds, with each epoch.

We also record the model reconstruction errors on each training batch. Taking the weighted RAE training process as an example, our training data contains 4,770 samples, 10 samples for each batch. The reconstruction errors on each batch are shown in Figure 6. Because reconstruction errors on different batches in the same epoch are different, the curve in the figure is oscillatory in each epoch. Additionally, because the reconstruction errors on the same batch in different epochs are decreasing, the overall downward trend of the curve can be seen from the figure.

2) RESULTS OF EXPERIMENT1 FOR ANSWERING RQ1

First, we compute recall and precision on all tagged clone pairs in BigCloneBench with different thresholds and plot the P-R curves, as shown in Figure 7. In the figure, we use traditional_RAE to represent the unweighted RAE [14], weighted_RAE to represent our model, and random_RAE
to represent the RAE model, which is not trained and instead assigned random neural network weights.

Here, we can see that our model is more accurate than the unweighted RAE in precision and recall. However, we can also see that the precision is always greater than 96% because of the class imbalance in BigCloneBench (detailed in Section VII-A), and thus, we need more indicators to measure our method.

Second, we compute TPR (here, the same as recall) and FPR on all tagged clone pairs in BigCloneBench with different thresholds and plot the ROC curves, as shown in Figure 8. As we can see, in the analysis of samples, whether the label is positive or negative, our model is more accurate than the unweighted method.

Finally, as the false clone pairs tagged by BigCloneBench do not contain Type-1 and Type-2; thus, we compute FPR on all false clone pairs but compute TPR on true clone pairs of each clone type. We plot ROC curves for each clone type with different thresholds, as shown in Figure 9.

The weighted RAE exceeds the unweighted RAE in detecting clone pairs of arbitrary clone types except for Type-2. When detecting Type-2 clone pairs, the weighted RAE is slightly inferior to the unweighted RAE.

3) RESULTS OF EXPERIMENT2 FOR ANSWERING RQ2

Each function in the source repository bcd_reduced is encoded into a vector with 296-dimensions by our model, and we obtain a set of 785,438 vectors. Searching neighboring vectors in the set requires quadratic pairwise comparisons, which entails 308,456,033,203 comparisons.

On our computer with the configuration as shown in Section VII-C, if we use np.linalg.norm offered by the NumPy package in Python to compute the Euclidean distance of two vectors, every 10,000 comparisons take approximately 40 milliseconds and the overall time needs approximately 14 days. Obviously, this system is not practical for code clone detection.

We use the NSG algorithm to complete code clone detection (see Section VI), and for each vector, we identify 200 neighbors and then postprocess these candidate clone pairs. The time cost of each stage is shown in Table 5. Obviously, the overall time cost is acceptable. The last stage is slow because it needs to further use the Euclidean distance metric to screen the candidate clone pairs returned by an NSG search, the number of which is 157,087,600 and much fewer than 308,456,033,203. Although our model is trained in approximately 10 hours, we do not use the tagged functions in BigCloneBench as the training dataset. Thus, our model can also be applied to arbitrary Java projects with no need to train it again.

We also compare recall (see Section VII-A) to that of using our method directly on tagged clone pairs in BigCloneBench, as reported in Table 6. We can see that our method is scalable to the large-scale software system and retain the recall when the set threshold is small.

3. RESULTS OF EXPERIMENT2 FOR ANSWERING RQ2

Each function in the source repository bcd_reduced is encoded into a vector with 296-dimensions by our model, and we obtain a set of 785,438 vectors. Searching neighboring vectors in the set requires quadratic pairwise comparisons, which entails 308,456,033,203 comparisons. On our computer with the configuration as shown in Section VII-C, if we use np.linalg.norm offered by the NumPy package in Python to compute the Euclidean distance of two vectors, every 10,000 comparisons take approximately 40 milliseconds and the overall time needs approximately 14 days. Obviously, this system is not practical for code clone detection.

We use the NSG algorithm to complete code clone detection (see Section VI), and for each vector, we identify 200 neighbors and then postprocess these candidate clone pairs. The time cost of each stage is shown in Table 5. Obviously, the overall time cost is acceptable. The last stage is slow because it needs to further use the Euclidean distance metric to screen the candidate clone pairs returned by an NSG search, the number of which is 157,087,600 and much fewer than 308,456,033,203. Although our model is trained in approximately 10 hours, we do not use the tagged functions in BigCloneBench as the training dataset. Thus, our model can also be applied to arbitrary Java projects with no need to train it again.

3) RESULTS OF EXPERIMENT2 FOR ANSWERING RQ2

Each function in the source repository bcd_reduced is encoded into a vector with 296-dimensions by our model, and we obtain a set of 785,438 vectors. Searching neighboring vectors in the set requires quadratic pairwise comparisons, which entails 308,456,033,203 comparisons.

On our computer with the configuration as shown in Section VII-C, if we use np.linalg.norm offered by the NumPy package in Python to compute the Euclidean distance of two vectors, every 10,000 comparisons take approximately 40 milliseconds and the overall time needs approximately 14 days. Obviously, this system is not practical for code clone detection.

We use the NSG algorithm to complete code clone detection (see Section VI), and for each vector, we identify 200 neighbors and then postprocess these candidate clone pairs. The time cost of each stage is shown in Table 5. Obviously, the overall time cost is acceptable. The last stage is slow because it needs to further use the Euclidean distance metric to screen the candidate clone pairs returned by an NSG search, the number of which is 157,087,600 and much fewer than 308,456,033,203. Although our model is trained in approximately 10 hours, we do not use the tagged functions in BigCloneBench as the training dataset. Thus, our model can also be applied to arbitrary Java projects with no need to train it again.

We also compare recall (see Section VII-A) to that of using our method directly on tagged clone pairs in BigCloneBench, as reported in Table 6. We can see that our method is scalable to the large-scale software system and retain the recall when the set threshold is small.

If the set threshold is too large, which means that function pairs with large Euclidean distances between their vectors will be tagged as true clone pairs, directly measuring the model on the tagged clone pairs obtains much higher recall, but the method based on NSG does not change considerably.
The reason is that the method based on NSG reports clone pairs by screening the candidates returned by the NSG search, the number of which depends on the number of neighbors that we search for each query. Therefore, to retain the recall when the set threshold is large, we need to identify more neighbors for each query vector during the NSG search, but it will cost more time in the third and the fourth stages (see Table 5).

The performance of weighted RAE on BigCloneBench using different evaluation methods. All data are represented in percentage. In evaluation method 1, we combine our model and the NSG algorithm to work directly on the large-scale source repository bed_reduced, and we estimate precision using the approach proposed by Svajlenko et al. [20]. We cannot compute FPR, which has no estimation approach provided in the aforementioned work because BigCloneBench does not tag all clone pairs in bed_reduced. In evaluation method 2, our model is directly applied to the clone pairs tagged by BigCloneBench. These two evaluation methods are detailed in Section II-C.

<table>
<thead>
<tr>
<th>Performance indicators</th>
<th>threshold=0</th>
<th>0.05</th>
<th>0.1</th>
<th>0.15</th>
<th>0.2</th>
<th>0.25</th>
<th>0.3</th>
<th>0.35</th>
<th>0.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2 Recall</td>
<td>1.78/1.78</td>
<td>89.12/89.12</td>
<td>94.63/94.63</td>
<td>96.34/96.34</td>
<td>97.16/97.18</td>
<td>97.65/97.83</td>
<td>98.65/99.86</td>
<td>98.65/99.46</td>
<td>98.65/99.89</td>
</tr>
<tr>
<td>VST3 Recall</td>
<td>1.29/1.29</td>
<td>78.79/78.79</td>
<td>89.91/89.91</td>
<td>93.50/93.50</td>
<td>98.11/98.49</td>
<td>99.42/99.80</td>
<td>98.95/99.81</td>
<td>98.95/99.93</td>
<td>98.97/100.00</td>
</tr>
<tr>
<td>ST3 Recall</td>
<td>0.04/0.04</td>
<td>20.88/20.88</td>
<td>46.55/46.55</td>
<td>61.15/61.26</td>
<td>70.50/72.63</td>
<td>74.18/81.73</td>
<td>75.80/91.19</td>
<td>75.93/95.55</td>
<td>75.93/98.52</td>
</tr>
<tr>
<td>MT3 Recall</td>
<td>0.15/0.15</td>
<td>0.38/0.38</td>
<td>3.46/3.46</td>
<td>12.51/12.61</td>
<td>26.34/31.53</td>
<td>33.62/53.18</td>
<td>36.60/71.51</td>
<td>37.35/84.60</td>
<td>37.42/91.71</td>
</tr>
<tr>
<td>T4 Recall</td>
<td>0/0</td>
<td>0/0</td>
<td>0.03/0.03</td>
<td>0.47/0.48</td>
<td>2.47/3.28</td>
<td>4.46/10.52</td>
<td>5.38/22.35</td>
<td>5.60/37.17</td>
<td>5.64/52.66</td>
</tr>
<tr>
<td>FPR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-0.09</td>
<td>-0.52</td>
<td>-2.09</td>
<td>-5.54</td>
<td>-11.80</td>
</tr>
</tbody>
</table>

FIGURE 9: ROC curves for each clone type from Type-1 to Type-4. For Type-1 and Type-2, both reach the highest. The weighted RAE exceeds the unweighted RAE in the accuracy of detecting clone pairs of each type from Very Strongly Type-3 to Weakly Type-3 (Type-4). In addition, from Type-1 to Type-4, the areas under the curve become smaller, which means that the detection abilities of the two models both decrease because the clones are increasingly difficult to detect.
However, we do not need to set the threshold too large because setting the threshold too large will increase FPR considerably, which increases the false positives and is not acceptable.

4) DISCUSSION

We compare the detection performance of our method with those of publicly available clone detection tools, namely, Nicad [8], Deckard [10], SourcerCC [11], and CloneWorks [36]. We also compare our method with clone detection tools based on deep learning, including Oreo [24], DeepSim [28], CCLearner [26] and CDLH [27]. Because these tools use two different evaluation methods (see Section II-C), we discuss the comparative results for the two classes separately.

We set our threshold as 0.3 because in this configuration, the FPR on tagged clone pairs of our model is 0.055; this value is a good indicator, which means our model will not produce many false positives.

To compare with DeepSim and CDLH, we apply our model directly to the tagged clone pairs and compute the precision, recall, and F1 value. Their measurements are taken from the reported values in the DeepSim paper, and the comparison results are as follows. Regarding the precision, our method achieves 99%, CDLH obtains 92%, and DeepSim achieves 97%. Because of the class imbalance of BigCloneBench (see Section VII-A), we think that they lack the FPR measurements. For recall, our method obtains 25%, CDLH achieves 74%, and DeepSim obtains 97%. The recall of our method is lower than those of the other methods; we believe this difference is mainly because those methods use a supervised learning method, whereas ours does not. Nevertheless, our method does not need labeled data.

To compare with SourcerCC, Nicad, CloneWorks, CCLearner and Oreo, we use our model directly on the source repository bc_dreduced and compute measurements according to the computational formulas in BigCloneBench’s paper [20]. The measurements of CCLearner are from its paper, and the measurements of the remaining detectors are from the Oreo paper. The authors of Oreo claim that Deckard cannot be evaluated directly on the source repository, and they quoted the performance measurements of Deckard from another paper. All the results are shown in Table 7, and we use the T1, T2, VST3, ST3, MT3 and T4 to signify the Type-1, Type-2, Very Strongly Type-3, Strongly Type-3, Moderately Type-3 and Type-4 (Weakly Type-3) respectively.

The precision of the other detectors is calculated via random sampling when sample strength is 400 [24], while our precision is estimated using the approach proposed in the BigCloneBench paper [20].

For Type-1, the recall of our method is 99.99% and not 100% because the clone pair, with one function’s id being 3,516,892 and the other function’s id being 22,648,747, is classified as Type-1 by BigCloneBench but not truly a Type-1 clone pair according to our manual verification. It is incorrectly masked by BigCloneBench and just a little noise.

As we can see, our method is better in detecting code clones of MST3 and T4, but slightly inferior in detecting code clones of T2, VST3, and ST3 because the set number of neighbors returned by the NSG search limits our performance. Without this limit, our method can obtain the recalls in detecting clone pairs of T1, T2, VST3, ST3, MT3 and T4 respectively as 100%, 99.06%, 99.81%, 91.19%, 71.51%, and 22.35% (see Table 6 where the set threshold is 0.3). However, we must use the NSG search to complete a fast code clone detection and decrease the time cost from original 14 days to 33 min (see Table 5). If we return more neighbors during the NSG search, we can obtain better recall indicators but the time cost of the third and fourth stages in Table 5 will also increase, which is not what we want to see.

Finally, we can conclude that our method owns a good performance compared with other clone detection tools.

VIII. THREATS TO VALIDITY

BigCloneBench is one of the few famous public evaluation benchmark for code clone detection and we evaluate our method on it comparing the evaluation results with that of many other clone detectors. However, this benchmark contains only Java programs and the effectiveness of our method when working on other programming languages is still yet to be assessed. Besides, the adaptation of our method to other languages is possible but it requires careful consideration of proposing some rules to transform the AST to a full binary tree that is a necessary for our model. Moreover, our method will not work for the code snippet that is not complete enough for parsing its AST.

Deep learning-based methods face the challenge of adjusting hyper-parameters, and our model is no exception. We mitigate this risk by performing sufficient replicates to

<table>
<thead>
<tr>
<th>Approaches</th>
<th>T1</th>
<th>T2</th>
<th>VST3</th>
<th>ST3</th>
<th>MT3</th>
<th>T4</th>
<th>Precision (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oreo</td>
<td>100</td>
<td>99</td>
<td>100</td>
<td>89</td>
<td>30</td>
<td>0.7</td>
<td>89.5</td>
</tr>
<tr>
<td>CCLearner</td>
<td>100</td>
<td>98</td>
<td>98</td>
<td>89</td>
<td>28</td>
<td>1</td>
<td>93</td>
</tr>
<tr>
<td>SourcerCC</td>
<td>100</td>
<td>97</td>
<td>93</td>
<td>60</td>
<td>5</td>
<td>0</td>
<td>97.8</td>
</tr>
<tr>
<td>CloneWorks</td>
<td>100</td>
<td>99</td>
<td>98</td>
<td>93</td>
<td>3</td>
<td>0</td>
<td>98.7</td>
</tr>
<tr>
<td>Nicad</td>
<td>100</td>
<td>99</td>
<td>98</td>
<td>93</td>
<td>0.8</td>
<td>0</td>
<td>99</td>
</tr>
<tr>
<td>Deckard</td>
<td>60</td>
<td>58</td>
<td>62</td>
<td>31</td>
<td>12</td>
<td>1</td>
<td>34.8</td>
</tr>
<tr>
<td>Our Method</td>
<td>99</td>
<td>98.65</td>
<td>98.95</td>
<td>75.80</td>
<td>36.60</td>
<td>5.38</td>
<td>99.62</td>
</tr>
</tbody>
</table>

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2938825, IEEE Access
choose the hyper-parameters.

IX. CONCLUSION AND FUTURE WORK

In this paper, we introduce a novel approach for fast code clone detection based on weighted RAE. It is a combination of deep learning and information retrieval. We use the TF-IDF model to weight different nonterminal types in AST and make RAE more accurate by amplifying the contributions of more important nodes to the final vector representations of programs. Detailed experiments on the standard benchmark BigCloneBench confirmed its effectiveness. Our proposed method is scalable to large-scale software systems and can return clone results in a relatively short time. Besides, our method is easier to migrate to a new application without labeled data. In the future, we will incorporate a supervised learning method to further improve the accuracy of detecting code clones.

ACKNOWLEDGMENT

The authors thank Martin White for offering considerable help with ast2bin, based on which much of our work improved, and thank Cong Fu and Jeffrey Svaženko for patiently answering some questions about NSG and BigCloneBench.

REFERENCES


**JIE ZENG** was born in Xinyang City, Henan Province, China in 1993. He received the B.S. degree in software engineering from Zhengzhou University, in 2013 and the M.S. degree in computer science from State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, in 2016. He is currently a Ph.D. candidate in the Naval University of Engineering, Wuhan, China. His research interests include software analysis, software defect mining, and machine learning.

**XIAOWEI LI** received the M.S. degree in management from Beihang University, in 2011. She currently works for Jinghang Research Institute of Computing and Communication, Beijing, China. Her research interests include software testing and software engineering.

**KERONG BEN** was born in 1963. He is currently a senior member of China Computer Federation and a PhD supervisor in the Naval University of Engineering. His main research interests include software quality assurance and artificial intelligence.

**XIAN ZHANG** was born in 1990 and received the Ph.D. degree from the Naval University of Engineering, Wuhan, China, in 2019. His research interests include software quality assurance, machine learning, and operational research.