TECCD: A Tree Embedding Approach for Code Clone Detection

Yi Gao*, Zan Wang*, Shuang Liu*, Lin Yang*, Wei Sang* and Yuanfang Cai†
* College of Intelligence and Computing, Tianjin University, † Drexel University

***Abstract*—Clone detection techniques have been explored for decades. Recently, deep learning techniques has been adopted to improve the code representation capability, and improve the state-of-the-art in code clone detection. These approaches usually require a transformation from AST to binary tree to incorporate syntactical information, which introduces overheads. Moreover, these approaches conduct term-embedding, which requires large training datasets. In this paper, we introduce a tree embedding technique to conduct clone detection. Our approach first conducts tree embedding to obtain a node vector for each intermediate node in the AST, which captures the structure information of ASTs. Then we compose a tree vector from its involving node vectors using a lightweight method. Lastly Euclidean distances between tree vectors are measured to determine code clones. We implement our approach in a tool called TECCD and conduct an evaluation using the BigCloneBench (BCB) and 7 other large scale Java projects. The results show that our approach achieves good accuracy and recall and outperforms existing approaches.**

***Index Terms*—Code Clone Detection; AST; Skip-gram;**

I. Introduction

Code clone, which usually refers to copying-and-pasting a piece of code with some modifications, is a common coding practice in software engineering. While code clones may increase the development speed, they also have adverse impacts [16], [50], such as introducing maintenance problems [49] and even bugs which are hard to detect [41]. Although it has been reported that clones may not always be harmful [32], it is worthwhile for developers to detect code clones. For example, code clones have been shown to be useful for software refactoring [47], [48], and source code comprehension [49], [55].

Code clones are classified into 4 types based on the degree of changes in the code. Type-1, Type-2 clones reflect textural changes; Type-3 clones capture syntactic similarities in program code [17]; Type-4 clones refer to semantic (functional) similarities [23]. Various approaches [30], [28], [35], [23], [51], [56], [40], [66] have been proposed to detect code clones. These approaches are mainly classified into 5 categories [57], i.e., text-based method, token-based method, tree-based method, graph-based method and metric-based method. More details on the related works can be found in section VI.

Type-1 and Type-2 clone detections [28] [30], [21] have been well studied. It has been reported that there are more Type-3 clones than other types in the repositories [58]. Therefore, in our work, we focus on Type-3 clone detection. The most popular Type-3 clone detection techniques [13] include token-based [28], [61], Tree-based [29] techniques. Recently machine learning-based [69], [40] techniques have been proposed to detect Type-3 clones.

Token-based methods [28], [61] parse source code into tokens, then some kind of index is built for searching similar blocks [61]. Tree-based methods [29], [16] utilize parse trees or Abstract Syntax Trees (AST) to capture code structure information. After that, subtrees are searched to detect code clones. Token-based methods are efficient, but not very effective in detecting Type-3 clones [28], especially for large scale modifications. Tree-based methods, on the other hand, have demonstrated better effectiveness in detecting Type-3 clones, but suffer from efficiency problems [29] , and thus are difficult be applied to large projects [54].

Recently, with the rapid development of machine learning techniques, there have been approaches [69], [40] proposed to use deep learning methods to conduct code clone detection. However, these approaches learn representations from tokens or terms, they cannot detect the cases where the source code is heavily modified textually, but the main structure remains, as shown by the code clone example in Fig. 1. The example is obtained from the `jEdit` project [1]. The `PhysDown` function scrolls down the cursor by the given number of lines. The `PhysUp` function scrolls up the cursor. Similar cases where similar functionalities (with similar implementations) are copied and modified, are common practices in software development. We've observed many such cases in our experiment. There are in total 12 lines (highlighted in red in Fig. 1) of modifications from the method in Fig. 1a to obtain the method in Figure. 1b. The scale of the changes (in terms of insertion, deletion and modifications) is more than half of the code size. However, if we check the main structures of the two pieces of code as squared in blue rectangles in Fig. 1), they are highly similar. Both methods have 4 `if-else` control structures, each of which controls similar program logic. This is a typical Type-3 clone. However, it cannot be detected by state-of-the-art clone detection tools, such as CCAligner [66], CCLearner [40] and Nicad [56].

In this paper, we aim to address this problem. Our goal is to improve the effectiveness (measured in terms of precision and recall) of Type-3 clone detection, and in the meantime, maintain the efficiency of detection. Inspired by the successful application of estimating word representations in vector space, we propose to convert ASTs into vectors and detect clones based on the distances between those vectors.

White et al. [69] proposed to map code fragments into

Shuang Liu is the corresponding author

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00025

```
1.  void physDown(int amount, int screenAmount){
2.      int currentPhysicalLine = getPhysicalLine();
3.      int currentScrollLine = getScrollLine();
4.      if(Debug.SCROLL_DEBUG){
5.          Log.log(Log.DEBUG,this,"physDown() start:"
            + currentPhysicalLine + ':' + currentScrollLine);
6.      }
7.      setSkew(0);

8.      if(!getDisplayManager().isLineVisible(currentPhysicalLine)){
9.          int lastVisibleLine =
              getDisplayManager().getLastVisibleLine();
10.         if(currentPhysicalLine > lastVisibleLine)
11.             setPhysicalLine(lastVisibleLine);
12.     }else{
13.         int nextPhysicalLine = getDisplayManager().
            getNextVisibleLine(currentPhysicalLine);
14.         assert nextPhysicalLine > 0;
15.         amount -= nextPhysicalLine - currentPhysicalLine;
16.         moveScrollLine(getDisplayManager().
            getScreenLineCount(currentPhysicalLine));
17.         setPhysicalLine(nextPhysicalLine);
18.     }
19.     if(screenAmount < 0)
20.         scrollUp(-screenAmount);
21.     else if(screenAmount > 0)
22.         scrollDown(screenAmount);
23. }
```

(a) Type-3 clone example (source)

```
1.  void physUp(int amount, int screenAmount){

2.      if(Debug.SCROLL_DEBUG){
3.          Log.log(Log.DEBUG,this,"physUp() start:"
            +getPhysicalLine()+ ':' + getScrollLine());
4.      }
5.      setSkew(0);
6.      int currentPhysicalLine = getPhysicalLine();
7.      if(!getDisplayManager().isLineVisible(currentPhysicalLine)){
8.          int firstVisibleLine =
              getDisplayManager().getFirstVisibleLine();
9.          if(currentPhysicalLine < firstVisibleLine)
10.             setPhysicalLine(firstVisibleLine);
11.     }else{
12.         int prevPhysicalLine = getDisplayManager().
            getPrevVisibleLine(currentPhysicalLine);

13.         amount -= currentPhysicalLine - prevPhysicalLine;

14.     }
15.     if(screenAmount < 0)
16.         scrollUp(-screenAmount);
17.     else if(screenAmount > 0)
18.         scrollDown(screenAmount);
19. }
```

(b) Type-3 clone example (target)

Fig. 1: Type-3 clone example

continuous-valued vectors with machine learning techniques to capture "context" information in the source code. This approach shares similar idea with our approach. However, the vector embedding is conducted on lexical level which makes it dependent on project-specific keywords, and a large training dataset is required to mitigate this effect. Moreover, time consuming operations such as conversion from a parse tree to a binary tree, and RNN-based tree encoding are conducted.

The basic idea of our work is to use AST to capture program structure information, then convert AST into continuous valued vectors through tree embedding. After that, a vector distance is calculated to measure the similarity of methods. We adopt a light-weight embedding approach to obtain the AST vector from its node vectors, which is more efficient compared to RNN-based approaches [69]. Unlike existing approach [69] which uses project-specific terms to train deep learning models, our work does not use lexical information in the learning process, which makes the trained node vector representation generally applicable across projects.

The contributions of our work are three-fold.

- We propose a tree embedding approach to detect Type-3 clones. Our approach is light-weight compared to existing approaches that adopt embeddings to represent codes.
- We implement our approach in a tool called TECCD , which is available in [2].
- We conduct thorough evaluations on 6 large-scale Java projects and the BigCloneBench. We compare our approach with 3 state-of-the-art approaches on all those datasets. The evaluation results show that our approach outperforms the state-of-the-art methods in clone detection effectiveness. Our approach also shows a comparable detection time with existing approaches, and is much more efficient than existing tree-based approach.

The remaining of the paper is organized as following. Section II provides preliminary knowledge related to our work. Section III introduces our approach in detail. We report the implementation and evaluation of our approach in detail in Section IV. Section VI discusses related works for code clone detection. Finally, we conclude our paper in Section VII.

II. Preliminary

A. Category of Code Clones

Code clone generally refers to copying-and-pasting a piece of code with a certain level of modifications. In literature, cloned code has been further divided into four types [57], [54]. We verbally describe the four types of clones that are widely adopted in the community as follows:

- Type-1 Clone: Code fragments are identical except for small changes in white space, layout, and comments.
- Type-2 Clone: Code fragments are structurally and syntactically identical except for variations in identifiers, literals, types, layout and comments.
- Type-3 Clone: On the basis of Type-1 and Type-2, there are other changes to the copied segment, such as modification, insertion or deletion of statements.
- Type-4 Clone: Code fragments are semantically similar, i.e., perform similar functions, but are syntactically different.

Among the 4 types of clones, Type-1, Type-2, Type-3 are syntax-based code clone. Type-4 is semantic-based code clone, which means the codes perform similar funcitonalities but have different syntactical structures [59]. To the best of our knowledge, there are no quantitative or formal definitions of different clone types. Especially for Type-3 clones, to what extend the modifications are allowed is not clear. Expert evaluations are commonly adopted to decide real clone pairs.

B. Skip-gram

Skip-gram [45] is a popular language model based on neural network to conduct word embedding task, which maps a word to a low-dimensional vector. The resulting word vectors contain rich semantic information in terms of context. Each word vector in the Skip-gram model represents the distribution of the context. Due to the efficiency in model training and the ability to capture rich semantic information [53], the Skip-gram model has been widely adopted in Natural Language Processing (NLP) tasks. Compared to natural language, programming languages are not ambiguous and has less context changes. Therefore, we adopt the Skip-gram model to obtain AST node-vector embeddings for Java programs.

C. Node2vec

Inspired by the word-embedding techniques in the natural language processing domain, Grover and Leskovec [26] proposed an approach called node2vec, in which a mapping of nodes to a space of features, which maximize the likelihood of preserving network node neighbourhoods, is learnt. In this way, node2vec represents nodes in a network graph as vectors (of continuous features). Different from sentences, network does not show a linear structure. Therefore the sliding window technique, used to capture context in natural language, is not applicable to network structures. In classical searching strategies, Breath-First-Search (BFS) and Depth-First-Search (DFS) are commonly used to capture two extreme scenarios. DFS captures the homophily equivalence relationship and BFS captures the structural relationship. A network usually exhibit both behaviours and thus a proper combination of BFS and DFS is necessary to capture the context of each node. To capture the "context" information in a network, node2vec proposed to use random walk, which tries to balance the searching priority between BFS and DFS through two proposed parameters (Return and In-out parameter), to capture the diverse neighbourhoods, and thus "context" information of a network graph. The evaluation shows that it can capture the homophily and structural relationship of nodes in a graph.

III. Our Approach

In this section, we present the main steps of our clone detection approach. The initial idea is to detect code clones based on code structure information, and at the same time, avoid the time consuming tree-based searching algorithms. To achieve this goal, we first convert source code into an Abstract Syntax Tree (AST), which contains code structure information. After that, we map an AST to a vector based on deep learning techniques and compare the Euclidean distance [19] of the vectors to detect code clones. We also implement our approach in a the clone detection tool called TECCD , which is available on our GitHub project [2]. Following existing clone detection approaches, such as NICAD [56], CCLearner [40] and CloneManager [33], our approach focuses on detecting code clones at the method level since method is the basic unit of software implementation and reuse.

Fig. 2 illustrates the overview of our approach, which includes a pre-processing step and three main steps. The pre-processing step generates a node-vector dictionary for AST nodes. The dictionary acts as the basis to convert an AST into a vector. The main process starts with generating an AST for each method, based on which a set of nodes for the AST is obtained (Step 1); then we convert the AST into a vector through sentence embedding [14] (Step 2); lastly we compute and compare the distance of the vectors to detect code clones (Step 3). In the remaining of this section, we introduce each of the above steps in detail.

A. Pre-processing

The details of the pre-processing step are shown in Fig. 2. First, we use ANTLR [3] to generate one AST for each method in the training corpus. Then we conduct a filtering step to remove the stop nodes, which we will describe in detail shortly. After the filtering step, we conduct a random walk [26] on each AST to obtain a set of node sequences for each AST. The idea is motivated by node2vec [26], in which a fix-length random walk is used to capture the neighbourhood relations of nodes in a graph. The obtained node sequences capture the structure information of the AST. Lastly, we use the node sequences obtained for all methods in the corpus to train a skip-gram model [4] and generate a node-vector dictionary, where the node name is used as index for querying the corresponding vector.

Obtain context information One of our main ideas is to convert an AST into a vector to improve the efficiency of the tree-based clone detection process. In particular, the Skip-gram [4] algorithm is adopted for the node embedding process. Skip-gram is successfully applied in natural language processing tasks to capture the context information of a sentence, where each sentence is truncated into a set of fixed length phrases (based on the windows size) and each word in a sentence is converted into a vector. Our problem is different from the task in natural language, where natural language sentences are

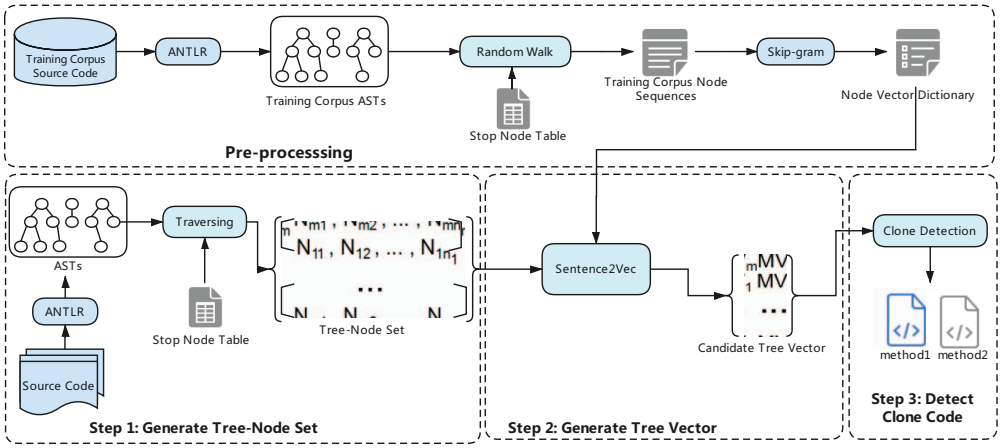


Fig. 2: The overall approach

in sequential structures and the context can be represented in sequential format, i.e., phrases. However, AST is not a sequential structure. Therefore, to enable embedding on an AST, we need to be able to obtain the "context" information of an AST in sequential format effectively.

Traditional approaches to traverse a tree structure include Breadth-First Search (BFS) and Depth-First Search (DFS). Each of the searching methods can provide a sequential representation of a tree. BFS produces sequences in which brother nodes are neighbours with each other, while DFS produces sequences where parent-children nodes are next to each other. Let's take Fig. 3 as a running example. Fig. 3a is a Java method, Fig. 3b is the AST generated for the method[1]. Fig. 3c shows example node sequences obtained through random walk on the AST of Fig. 3b. In Fig. 3, the nodes `expression`, `statement`, which are brother nodes of each other[2], are children of the `IfThenStatement` node. This structure can be obtained from one step derivation of the Java grammar rule `ifThenStatement` → `'if' '(' expression ')' statement`. BFS is good at capturing the local structures of a complex expression/statement, such as the `ifThenStatement`. While DFS is good at capturing relatively simple, yet sequential derivations. For example, in Fig. 3b, the node sequence `expression, assignmentExpression, postfixExpression, literal` (in the dotted rectangle) represents the stepwise derivation (according to ANTLR Java 8 Grammar) of a method parameter and all (single) argument occurrences have the same sequence (parsed with ANTLR). This sequence acts as a context information for the AST structure.

From the above analysis, we observe that both BFS and DFS provide sequences that are useful to capture the structure/context of an AST. However, none of them can provide sufficient context information. Therefore, we conduct a random walk [26], which combines both BFS and DFS searching order, on an AST to obtain the context information. Inspired by the random walk algorithm proposed in [26], in which a fix-length random walk is used to capture the neighbourhood relations of nodes in a graph, we propose to conduct a random walk on AST. Since the original algorithm is developed for graphs, it cannot be directly applied to AST. The difference is that for trees, there is no explicit edges between brother nodes and no circles either. However, the random walk algorithm in [26] requires explicit edges to conduct BFS-like traversal. Therefore, in our approach, we modify the AST slightly to add directed edges between brother nodes. For example, the dashed arrow from `blockStatement` to `blockStatement` in Fig. 3b is the added transition to enable BFS-like traversal with random walk. We modify the random walk algorithm in [26] to accommodate the tree structures.

Random Walk Formally, given a source node *u*, each node in a fixed length random walk (starting from *u*) is generated with the following distribution [26]:

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & if(v,x) \in E \\ 0 & otherwise \end{cases} \quad (1)$$

where l is the fixed length, $1 \leq i \leq l$, $c_0 = u$, v is the current node and x is the next node to visit. E is the set of edges in the tree. We modify the 2^{nd} order random walk transition probability (from [26]) to adjust to tree structures. Formally, $\pi_{vx} = \alpha_{pq}(t,x)$ and

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & if d_{tx} = 1 \\ \frac{1}{q} & if d_{tx} = 2 \end{cases} \quad (2)$$

Consider a random walk has just traversed edge *(t, v)* and is currently at node *v*. The algorithm decides the next node *x* to visit based on the value of d_{tx}, which represents the shortest path distance between nodes *t* and *x* and the value must be one of $\{0, 1, 2\}$. Different from graphs, there is no such cases where $d_{tx} = 0$ (which indicates a backward traversal on the graph) in a tree. Therefore, we only consider the cases where $d_{tx} = 1$ or $d_{tx} = 2$. $d_{tx} = 1$ indicates that *t* and *x* are parent-child relation, which also indicate that *v* and *x* are brother

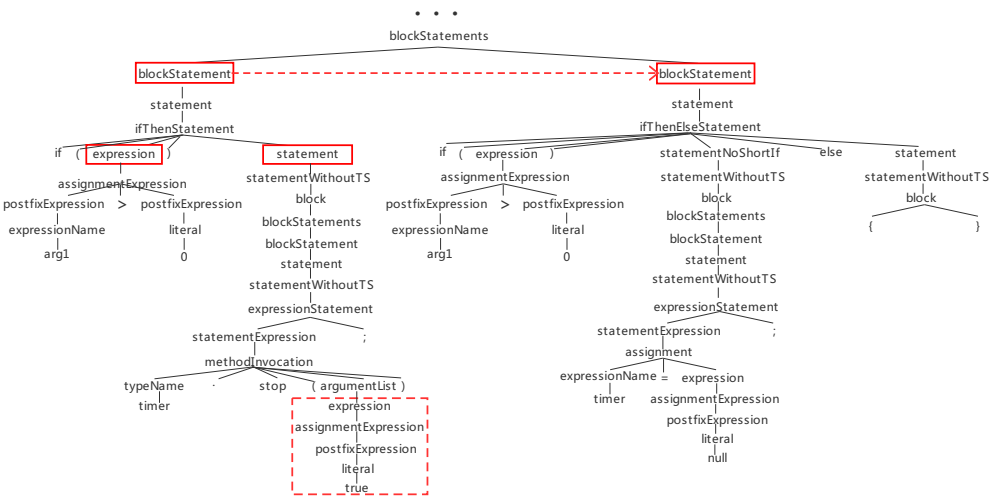
[1]We remove the method declaration for the simplicity of illustration.
[2]Our approach does not consider leaf nodes.

```
public void source(){
    if(arg1>0){
        timer.stop(true);
    }
    if(arg2>0){
        timer = null;
    }else{
    }
}
```

(a) An example Java method

```
[blockStatements, blockStatement, blockStatement, statement,
 ifThenElseStatement, expression, statementNoShortIf,
 StatementWithoutTrailingSubstatement, block,
      blockStatements];
[blockStatement, statement, ifThenStatement, expression,
 statement, StatementWithoutTrailingSubstatement, block,
 blockStatements, blockStatement, statement];
[statement, ifThenStatement, expression,
     assignmentExpression,
 postfixExpression, literal]
```

(c) Random Walk Node Sequences



(b) An example AST (for the Java method)

Fig. 3: A running example

nodes. $d_{tx} = 2$ indicates that v and x are parent-child relation. The parameters p, q controls the likelihood of conducting BFS-like and DFS-like searches. In particular, p controls the probability of conducting BFS-like search, and q controls the probability of conducting DFS-like search. In our experiment, we do not have preference on either BFS-like or DFS-like search, therefore we set $p = q$.

Fig. 3c illustrates three example node sequences (with maximum length set to 10 for clear illustration) generated for the AST in Fig. 3b through our random walk algorithm[3]. Each generated node sequence contains a mixed brother-brother, parent-child relations. In this way, our approach converts a tree structure into a sequential structure, and meanwhile maintains (to some extend) the original tree structure information. After the random walk on an AST, a set of node sequences are obtained , which represent the context information of intermediate nodes in an AST. We take the obtained node sequences as input to train the skip-gram model.

We chose 7 popurlar open source Java project, i.e., JDK1.8 [5], Ant 1.10.1 [6], Commons Lang-3.3.7 [7], jEdit 5.4.0 [1], JDK 1.2.2 [5], Maven 3.5.0 [8] and OpenNLP 1.8.1 [9], as training corpus projects to train a Skip-gram model and obtain the node-vector dictionary. These projects all have large code bases and cover a large diverse of different coding styles. Note that the node-vector dictionary generation is one-time effort, i.e., once the dictionary is generated, any Java project is able to reuse the dictionary. This is achievable since our approach only considers the non-leaf nodes of an AST for the vector generation. Recall that the non-leaf nodes of an AST capture structure information of the program, and are project-independent. The project-specific information, such as the variable name, is represented as leaf node in AST and is not considered for node vector generation in our approach.

Results of Context Information Capturing We explicitly analysed the results of node embedding through context information obtained through random walk on AST. We are interested in understanding how random walk on AST can capture the structural context of an AST. By a simple visualization using TSNE [27], we can observe that AST nodes with similar context (in terms of random walk node sequences) are embedded into vectors of small distances. Such examples include the `tryStatement` (166), `catches` (167), `catchClause` (168), `catchFormalParameter` (169),

[3]The stop nodes are removed

`catchType` (170) and `finally` (171). These nodes always co-appear in the tryStatement subtree and thus sharing similar "context" (father-child, brother-brother) nodes. Therefore, the resulting node embedding vectors have small distances with each other. On the other hand, AST nodes with different contexts are embedded into vectors of larger distances. This results indicate that the node vectors obtained actually capture the context information of a tree. Therefore, we are able to use the obtained vectors to conduct structure-based clone detection.

Stop Nodes AST tends to have a huge number of nodes as introduced for proper parsing purposes and it is rather time consuming to process. ANTLR is designed to generate 237 types of nodes in total, among which there are intermediate nodes that appear frequently in AST, but do not differentiate the code structures. For example, the derivation sequence from an expression node to the specific term is `expression, assignmentExpression, postfixExpression,...` (as shown in Figure 3b). Nodes in this sequence always have similar context (random walk always provides the same sequences), and thus they have similar vectors after node embedding. However, these nodes are redundant and do not contribute to distinguish different tree structures. Through a manual checking on the parsing results of 7 large Java projects, we found that there are in total 55 such kind of intermediate nodes which are useful for parsing, but are not useful for differentiating code structures. We refer this kinds of nodes as the stop nodes[4], and filter these nodes in our approach to improve clone detection accuracy.

B. Step 1: Generate Tree-Node Set

The first step of clone detection is to generate a node set for a given Java method[5]. As shown in `Step 1` of Fig. 2, our approach takes the source code of a Java method as input. ANTLR [3] is used to obtain an AST for each input method. Then we conduct depth first search (DFS) to traverse the ASTs and obtain a tree-node set for each AST. During the traversing, the stop nodes are removed. For example in Fig. 2, each row, e.g., $\{N_{m1}, ..., N_{mn_m}\}$, is a tree-node set generated for the AST (and thus the corresponding method) indexed m. Suppose we have m methods in the project to be checked, then we obtain m such tree-node sets.

C. Step 2: Generate Tree Vector

In the second step, we generate a vector for each AST. Motivated by the works in NLP community, we adopt the sentence to vector method [14] to obtain a vector for an AST (method) from its node vectors. In our approach, the structure information is captured during the pre-processing phrase, i.e., while conducting node vector embedding. Therefore, we adopt a light-weight sentence embedding approach, in which no order information of the AST nodes is preserved.

[4]This naming is inspired by the stop words.

[5]Our approach operates on AST intermediate nodes. Therefore, the generated tree-node set represents the method information.

For each AST, the sentence to vector method [14] obtains a vector by applying sentence embedding. In sentence embedding, each node corresponds to a word in a sentence and each AST (composed of a set of nodes) corresponds to a sentence. The nodes are converted to vectors by looking into the node vector dictionary obtained from the pre-processing step. After the sentence embedding step, we obtain one vector for each AST. For example in Fig. 2, the set of nodes $\{N_{m1}, ..., N_{mn_m}\}$ for AST m are converted to the tree vector MV_m.

D. Step 3: Code Clone Detection

After step 2, we obtain one vector for each AST. The vector captures the structure information of the corresponding method. Moreover, since Skip-gram considers context information and encodes such information into the vectors. Therefore, contextually similar nodes have similar vector representations. Based on this property, we can detect clone pairs based on the Euclidean distance [19] of the vectors. In addition, in the process of vector comparison, we use Locality Sensitive Hashing (LSH) [20] algorithm to improve the efficiency of comparison. We determine whether two pieces of code are cloned based on the vector distance.

Since the vectors generated by the ASTs of exact clone (Type-1 clone) and parametric clone (Type-2 clone) are exactly the same, the Euclidean distance of the vectors is zero. For Type-3 clone, the code structure may change due to code fragments insertion, deletion and modification. Therefore, the Euclidean distance of the tree vectors may increase. We need to set a threshold of the Euclidean distance, below which the compared methods are identified as Type-3 clones. Since there is no quantitative definition on Type-3 clones that we can refer to decide the distance threshold, we set our threshold based on experiments with existing clone benchmarks. We manually labelled 7 large Java project, with more than 70 thousand lines of code. These labelled projects are used to tune the threshold for Type-3 clones. Through manual inspection on the results, we select a threshold 0.02 which achieves a good precision and recall as our base threshold. Vector distance below this threshold is identified as Type-3 clones.

When tuning the threshold, we observe that there are cases where the code is changed in a few lines, e.g., adding an if-else structure, but the number of AST nodes are changed by a large portion (especially if the original method has a small number lines of codes). The resulting sentence vectors may have a large distance. The reason is that the number of intermediate nodes generated for the if-else statements is large and they dramatically affect the obtained tree vector. To solve this problem, rather than using a fixed threshold value for vector distance comparisons, we propose to use an adjustable threshold value. The intuition is to add the information of the proportion of code changes at the token level as a factor (α) of the base threshold (θ_{base}). The factor (α) affects the base threshold (θ_{base}) in a way such that the effect of structure changes on small code pieces may be mitigated. The factor α is computed by accumulating the line based Levenshtein distance [39] (on source code).

TABLE I: Dataset Information

Data Set	# Files	# LOC	# Methods
JDK 1.2.2 [5]	109	17, 140	1, 028
OpenNLP 1.8.1 [9]	850	66, 291	3, 748
Maven 3.5.0 [8]	872	79, 840	5, 309
Ant 1.10.1 [6]	1, 205	138,505	12, 620
Commons Lang 3-3.7 [7]	311	75, 210	6, 404
jEdit 5.4.0 [1]	592	118,407	7,160
BigCloneBench [63]	32, 317	8, 836, 676	168, 965

TABLE II: Settings of Clone Detection Tools

Tool	Min LOC	Criteria for Clone
TECCD	min 10LOC	0.02
CCLearner	min 10LOC	$l_c \geq 0.98$
CCAligner	min 10LOC	60% similarity
NICAD	min 10LOC	70% similarity

IV. Implementation and Evaluation

We implement our approach in a tool called TECCD . We use ANTLR (version 4.7.1) [3] to obtain AST from source code. The Skip-gram model [4] is adopted to convert an AST node into a vector and the sent2vec [14] model is adopted to generate a vector for each method. Our tool is public available at our GitHub project [2].

A. Experiment Setup

Dataset Selection In order to verify the effectiveness and efficiency of our approach, we use 7 large-scale open source Java projects and the BigCloneBench [63] as experimental data sets. The detailed information of the datasets is shown in Table I. Each project in the dataset contains a few hundred files, with the number of methods ranging from one thousand to twelve thousands. We manually labelled all the clone pairs of the first 7 and make them a code clone benchmark, which is available [10]. The BigCloneBench (BCB) [63] is a manually crafted benchmark for (especially Type-3 and Type-4) clones and is a widely adopted benchmark for code clone detection methods. It consists of more than 3 million manually labelled clone pairs. BCB typifies the clones and measure their syntactical similarity. As there is no consensus on the minimum similarity of a Type-3 clone, BigCloneBench distinguishes Type-3 and Type-4 clones by dividing them into three categories, i.e., the clones that share at least 70% similarity at the statement level is regarded as strong Type-3 clones; the clones having 50-70%, less than 50% shared syntax similarity are regarded as moderate Type-3 clones, weak Type-3 (Type-4) clones, respectively.

Comparison Methods Selection In our experiment, we compare our tool TECCD with existing state-of-the-art clone detection tools on both effectiveness and efficiency. For effectiveness comparison, we choose NICAD [56], CCLearner [40] and CCAligner [66], which have demonstrated state-of-the-art performances in Type-3 clone detection. NICAD is a classic and effective text-based clone detection tool, which has been shown to be effective in detecting near-miss clones. CCLearner adopts deep learning method for clone detection, which share similar idea with TECCD . CCAligner is one of the newest clone detection tools, which has also shown to be effective to detect Type-3 clones, especially large-gap clones. Recently, RNN-based methods [69], [18] are proposed to conduct code clone detections. These methods share similar ideas with our approach and would be interesting to compare with. However, to the best of our knowledge, the source codes of those approaches are not public accessible and thus we are unable to compare with those methods. We also explicitly evaluate the efficiency of our approach, and compare our tool with Deckard [29][6], which is a popular tree-based clone detection method, on a large dataset.

Experiment environment configuration We conduct our experiment on a machine with CentOS 7 operating system, Intel (R) Xeon (R) CPU $E5-2640v3$@ 2.60 GHz and 128 G memory. Before proceeding with the experiment, we need to configure each tool consistently. The detection granularity is chosen at the code method level. To make a fair comparison, we use 10 LOC as the minimum threshold to be considered as clone candidates for all 4 tools, following the the default setting for NICAD and CCAligner. We remark that, our tool TECCD can be configured to detect clones of any number of LOC. In our experiment, we use the default configurations of each tool to decide code clones. This is shown in the third column of Table II. Our method TECCD uses Euclidean distance to detect clones, and the base threshold is set to 0.02.

CCLearner implements a neural network with two hidden layers. It calculates the similarity between the number and type of tokens in two pieces of code, and then classifies them based on the trained neural network model. CCLearner set $l_c \geq 0.98$, which is the predicted probability, as the threshold to detect clones. Nicad is a text-based character-matching LCS algorithm to detect clones and 70% is the threshold as reported in the paper [56]. CCAligner is a token-based code cloning detection method, which can detect large-gap clone (a special Type-3 code clone). It uses $q = 6$ (the window size), $e = 1$ (the edit distance threshold) as the default setting for detecting clone pairs, and the similarity threshold is set to be 60%.

Based on our observations with the BigCloneBench [63], there are no Type-3 (and definitely Type-1 or 2) clones with 2.5 times or more differences in line numbers. Therefore, in our experiment, we avoid comparing such kind of code pairs. This is a commonly used heuristics in code clone detection methods [40].

B. Experimental Result

Clone Detection Effectiveness Evaluation We run all the clone detection tools with settings shown in Table II on all the datasets shown in Table I. The results on the effectiveness comparison of clone detection are reported in Table III and TableIV. In Table III, the first 2 columns show the data sets and the tools. columns $3-8$ show the results for Type-1 and Type-2 clone detection. Columns $9-14$ show the results for

[6]CCLearner has shown to outperform Deckard and thus we directly compare with CCLearner on effectiveness.

TABLE III: Comparison Results on 6 Projects

Data Set	Tool	Type-1, Type-2 Clone						Type-3 Clone						Time	
		# Pairs	TP	FP	Precision	Recall	F1	# Pairs	TP	FP	Precision	recall	F1	Preproc	Detection
JDK 1.2.2	TECCD	21	21	0	100	100	100	57	48	9	84	87	85	35s	3s
	Nicad	16	16	0	100	76	86	26	25	1	96	45	61		2s
	CCLearner	19	19	0	100	90	95	70	51	19	73	93	82		2s
	CCAligner	21	21	0	100	100	100	54	45	9	83	82	82		1s
Maven 3.5.0	TECCD	5113	5113	0	100	100	100	1924	1750	174	91	93	92	76s	15s
	Nicad	436	436	0	100	9	17	601	540	61	90	29	44		9s
	CCLearner	4437	4437	0	100	87	93	1548	1243	305	80	66	72		10s
	CCAligner	5021	5021	0	100	98	99	1558	1409	149	90	75	82		4s
OpenNLP 1.8.1	TECCD	613	613	0	100	100	100	1462	1374	88	94	95	94	79s	16s
	Nicad	200	200	0	100	33	50	648	632	16	98	43	60		8s
	CCLearner	584	584	0	100	95	97	1612	1271	341	79	87	83		9s
	CCAligner	604	604	0	100	99	99	1341	1204	137	90	82	86		3s
Ant 1.10.1	TECCD	443	443	0	100	100	100	1092	1015	77	93	91	92	148s	65s
	Nicad	141	141	0	100	32	48	519	495	24	95	44	60		16s
	CCLearner	439	439	0	100	99	99	1221	914	307	75	82	78		20s
	CCAligner	176	176	0	100	40	57	755	709	46	94	64	76		6s
jEdit 5.4.0	TECCD	224	224	0	100	100	100	1118	1028	90	92	92	92	193s	54s
	Nicad	124	124	0	100	55	71	845	626	219	74	56	64		16s
	CCLearner	209	209	0	100	93	96	1275	1050	225	82	94	88		8s
	CCAligner	197	197	0	100	88	94	1560	1089	471	70	97	81		5s
Commons Lang 3-3.7	TECCD	1440	1440	0	100	100	100	3180	2779	483	87	92	89	73s	24s
	Nicad	1007	1007	0	100	70	82	1098	1084	14	99	36	53		13s
	CCLearner	1348	1348	0	100	94	97	2859	2368	491	83	78	80		11s
	CCAligner	1200	1200	0	100	83	91	1652	1445	207	87	48	62		5s

TABLE IV: Results for the BigCloneBench

Tool	Recall						Precision
	T1	T2	VST3	ST3	MT3	WT3/T4	
TECCD	100	96	99	87	23	0	88
CCLearner	100	96	98	87	27	0	84
CCAligner	100	95	98	76	0	0	72
NICAD	99	95	98	92	0	0	86

Type-3 clone detection. *# Pairs* is the number of clone pairs detected. *TP, FP* are the number of true and false positives of the corresponding methods' detection results. *Precision*, *Recall* and *F1* are the precision, recall and F1-Score values (in terms of percentage), respectively.

From Table III we can see that, all of the tools can achieve 100% precision for Type-1 and Type-2 clones. TECCD performs the best in terms of recall. `CCLearner` and `CCAligner` also achieves good results. `Nicad` has the lowest recall on all datasets. Through manual checking on the experiment results, we find that `Nicad` has little tolerance on term changes. Therefore, it misses many Type-2 clones which have term changes. Type-1 and Type-2 clones have clear definitions and are relatively easier to detect. In general, TECCD is stable and more capable of detecting Type-1 and Type-2 clones. This is because Type-1 and Type-2 clones do not involve structure changes, and thus they always show a vector distance of 0. TECCD performs the best, `CCLearner` also performs well, and is more stable than `CCAligner` and `Nicad`.

For Type-3 clones, TECCD shows a consistently good performance on both precision and recall. `Nicad` tends to show relatively good precision, but bad recall. This is for the same reason as Type-1 and Type-2 clones explained before. `CCAligner` performs well on the `JDK` and `OpenNLP` datasets, but its performance is not stable on the other datasets, as indicated by the *F1-Scores*. Especially on the `Ant` and `Commons Lang` projects, the recall scores are 76% and 64%, respectively. the source code of these projects, we find that, the low scores are due to the default choices of edit distance $e = 1$ and windows size $q = 6$ to be the default setting, which shows the best performance. With such a setting, it cannot detect the cases where more than 2 lines are changed in the window block. There are many such cases in the `Ant` and `Commons Lang` projects. Therefore, `CCAligner` has a bad recall on this two datasets. `CCLearner` shows a stable, yet relatively worse performance compared to TECCD and `CCAligner`. The main reason may be that `CCLearner` uses program terms as training set, which is project-dependent and the effectiveness is affected by the training set. Moreover, only different types of tokens are counted, there is no structure information preserved. This may also be a reason for `CCLearner` to miss some clones, especially the large-gap clones.

To test how our approach generalize to other projects, we use the BigCloneBench [63], which is a manually crafted clone benchmark for all types of clones. The BigCloneBench clone pairs are selected from real programs without mutation operations, therefore, they represent real world code clones. The evaluation results with the BigCloneBench is shown in Table IV. The recall is calculated with the BigCloneEval [64], which automatically queries the BigCloneBench database for labelled results. The columns `T1, T2, VST3, ST3, MT3, WT3/T4` represent Type-1, Type-2, Strong Type-3, Medium Type-3, Weak Type-3/Type-4 clones, respectively. We can observe from the results, TECCD achieve near 100% recall for T1, T2 and VST3 clones, and TECCD outperforms all the other compared tools for these types of clones. Note that TECCD as well as the other three tools do not achieve a 100% accuracy, because the BigCloneBench labelling insufficienes, which has also been reported in other works [40]. NICAD performs well on ST3 clones, but poorly on MT3 clones. CCAlighner shows a worse performance than NICAD on ST3 and MT3 clones. TECCD and CCLearner shows good performance on ST3 clones. For MT3 clones, CCLearner and

TECCD both perform better than CCAlighner and NICAD, and CCLearner is a little better. Note that CCLearner is trained on more than $10K$ files taken from the BigCloneBench, but TECCD does not require training in the detection phrase. For precision, for the known reason that the BigCloneBench is not suffciently labelled, and it is infeasible to conduct a manual checking on all the clone pairs, we adopt the same approach as existing works [40] to sample a (statistically significant) subset of detected clone pairs from each tool and manually check the precision of the corresponding clone detection method. The sample size is 385, which is a statistically significant sample size with a 95% confidence level and $[-5\%, 5\%]$ confidence interval. The results are shown in the last column of Figure IV. TECCD achieves the best precision compared to other methods.

Clone Detection Efficiency Evaluation We report the execution time of the different clone detection methods in the last column of Table III. It takes TECCD 4.5 hours in total to generate the node dictionary, which is one-time effort. For TECCD , the `Preproc` column shows the time for AST generation, and the `Detection` column shows the time for the other steps of clone detection. For the other methods, since they are not tree based, there is no preprocessing required. For TECCD , majority time is spent on AST generation, which is a known drawback for tree-based clone detection methods. For detection, TECCD uses a bit more time than existing token-based approaches, but is comparable. Note that since our tool TECCD achieves higher recall, which means potentially more clone pairs are required to be processed. Therefore, it is reasonable that more time is consumed.

To further evaluate the efficiency of our tool, we also compare TECCD with the state-of-the-art tree-based clone detection tools `Deckard` [29] on a large folder of the BigCloneBench, which consists of more $2,688,875$ lines of code. The results show that TECCD uses $490min$ to finish the processing, while Deckard takes $2071min$, which is more than 4 times of TECCD . The evaluation results show that TECCD is compatible with non-tree-based method in terms of execution time and is more efficient than tree-based methods.

V. Threats to Validity

Context Information In our approach, we conduct random walk on AST to obtain the context information of nodes. Different from the Node2vec [26] approach, which only takes the context sequences of one graph as input, our approach takes the context sequences obtained from all ASTs as input to learn a node vector dictionary. When using the window to cut grams for training, there may exist cross-method grams, i.e., nodes from two different trees are put into one node gram. There are similar situations in processing natural language sentences, where words in a gram are obtained from the end of the previous sentence and the start of the next sentence. This kind of grams are taken as sentence-level context information in the natural language domain, while the neighbouring of two methods may not necessarily show context information in our approach. This may affect the training accuracy as this kind of node gram is not the real intended context.

Parameter Tuning There are certain parameters adopted in our approach, such as the vector dimensions, threshold for clone detection, and α, which are manually tuned. In our current setting, the parameters are tuned with some of the evaluation datasets, and the parameters which show best performance are used as the default setting. This is a commonly adopted practice for parameter tuning, although the parameters may be biased due to the dataset selection. To mitigate this effect, in the evaluation, we test our method (with tuned parameters) on the BigCloneBench, which is not involved in parameter tuning, and the evaluation results show that our method outperforms the other methods. Therefore, our parameter configuration is generalizable to some extend.

Evaluation with the BigCloneBench The BigCloneBench has known label deficiencies, i.e., it does not label all true clones. Therefore, the evaluation results on BigCloneBench is conducted on a partially labelled set, which may result in the measured recall be different from the actual recall. Moreover, the precision is calculated on a randomly sampled set of detected clones, which may introduce human bias or unintentional errors. We ask two master students to cross validate the results to decrease the potential risks. A crowdsourcing task may be a proper way to solve this issue and this is subject to our future improvement.

VI. Related Work

Token-based approach Token-based methods [41], [36], rely on lexical analysers to convert the source code into token sequences. Similar to the text-based approaches, the obtained token sequences are treated as strings. Then string matching algorithms are adopted to locate the lines where the cloned code exists. CCFinder [28] is one of the classic token-based detection approach. After converting the source code into token sequences, CCFinder uses suffix tree matching algorithm to detect similar token sequences. Livieri et al. [43] proposes a distributed method for large-scale code clone analysis. The method is implemented into a distributed version of CCFinder named D-CCFinder [43]. Basit et al. [15] proposes a simple and flexible tokenize method, and then uses a memory efficient suffix array for clone detection. SourcererCC [61] is a near-miss clone detector. SourcererCC is based on token, and it proposes the partial index algorithm and filtering heuristics to reduce the frequency and quantity of tokens required for detecting clones. Token-based detection algorithms can detect clones that involve different line structures, which cannot be detected by text-based detection algorithms. However, neither of the two methods fully consider the structure or semantic information of the code, and thus they can not detect most Type-3 and Type-4 clones, especially those with multi-line modifications and "dislocation".

Tree-based approach In tree-based code clone detection methods [71], [60], the source code is first parsed into parsing

tree [24] or abstract syntax tree (AST) [16]. Then tree matching algorithms are used to search similar subtrees, and the corresponding source code of the detected subtree is returned as the detected clone code. Feng et al. [22] introduces an AST-based method called AST-CC. The method first calculates the hash value of each node in AST. Then it compares the hash value of AST nodes to detect Type-3 clones. Deckard [29] is a tree-based clone detector, which approximates the structure information of ASTs by calculating certain characteristic vectors, and then use Locality Sensitive Hashing (LSH) to cluster the similar vectors. The code corresponding to the vectors that fall in the same cluster are considered clones. Wahler et al. [65] proposes to represent the source code as an AST in XML format. Because the generated trees contain the complete structure information of the source code, tree-based detection method shows capability in detecting both exact and near-miss clones. However, tree-based methods have high time and space overhead. Therefore, they are not suitable for analysing clones of large source code repositories.

Machine learning-based approaches With the rapid development of deep learning techniques [38], there have been approaches [52], [62], [67] proposed using machine learning methods to conduct code clone detection. White et al. [69] first proposes a deep learning-based clone detection method that locates clones by extracting features from program tokens. This method adopts RtNN [46] to map each term in a fragment to an embedding, and then transfers lexical information from the leaf to the root of the structure with RvNN [25]. The method requires converting an AST into a binary tree, which is time consuming. Moreover, a large training dataset is required since this method conduct embedding on terms (which are program specific). Wei et al. [67] designs a supervised deep feature learning framework, CDLH, to detect function similar clone. They apply traditional LSTM [72] on AST structure, and use the AST-based LSTM to generate real-valued representation of the source code, then the hash function is used to encode them as binary hash codes. Finally, they calculate the Hamming distance of hash codes to detect clone. Gemini [70] is the first neural network-based method to generate embedding of binary function. The binary code of the method is represented by ACFG and updated iteratively based on graph embedding network. Finally, the embedding vector of the method is obtained. Code2Vec [12] proposes a path-based attention model by studying the set of paths between one leaf node and another leaf node in AST as code representation. The learnt code representation is then used to predict method names across projects. CCLearner [40] is a token-based clone detection tool which adopts deep learning algorithms. It first classifies tokens in source code into eight categories and calculates the similarity score for each category of the two code snippets. Then it characterizes the relationship between the methods with the calculated similarity vectors. CCLearner extracts tokens from labeled cloned and non-cloned code sets, feeds them to a deep neural network (DNN) [11] to train a clone detection classifier. CCLearner requires a set of labelled cloned or non-cloned code, which is not easy to obtain. The accuracy of the trained model also depends on the training data. Due to the diversity of code clones, it is not easy to obtain a large labelled data set that cover different kinds of clones. Our approach adopts unsupervised training techniques, which does not require manually labelling and thus is not restricted by the training dataset. We adopt tree-based method to capture the structural information of the code. The experiment results shows that our method TECCD has higher precision and recall than CCLearner.

Other approaches Text-based code clone detection methods [30], [21], [31], [68] usually transfer source code into strings and then compare the characteristics of the corresponding strings. Murakami et al. [51] proposes a method detecting contiguous and gaped code clones based on Smith-Waterman algorithm, and implements the detection tool CDSW. NICAD [56] is a text-based hybrid clone detection tool. It first performs code normalization and then exploits the Longest Common Subsequence (LCS) comparison algorithm to identify clone. Text-based clone detection algorithms have high space-time efficiency. However, most text-based code clone detection techniques can only detect Type-1 clones [31], [21]. The graph-based code clone detection methods [23], [37] usually convert the source code into the form of graphs, such as Program Dependence Graph (PDG). The subgraphs with similarities above the detection threshold are selected as clone pairs. On the basis of the PDG, Liu et al. [42] adopts program slicing techniques to find isomorphic subgraphs. Graph-based detection technology can detect Type-4 clones. However, the algorithm for constructing a graph and matching subgraphs have high computational complexity. Metric-based code clone detection methods [35] [44] [34] extract a set of metrics from program source code, and then detect code clone based on the similarity of the metric values. Metric-based detection methods are efficient. However, the metric usually can only represent a part of program information, which makes the accuracy of the detection methods relatively low.

VII. Conclusion

We introduce a tree embedding technique to conduct code clone detection. Our approach first conducts tree embedding to obtain a node vector for each intermediate node in the AST. The node embeddings capture the context/structure information of ASTs. Then we compose a tree vector from its node vectors using a lightweight method. Lastly Euclidean distances between tree vectors are measured to determine code clones. We implement our approach in a tool called TECCD and conduct an evaluation with 7 large Java projects as well as the BigCloneBench (BCB). The results show that our approach outperforms existing approaches on accuracy and recall and is comparable with existing approaches in performance.

Acknowledgment

The paper is supported by National Science Foundation 61872263, 61802275, U1836214, and Tianjin Science and Technology Committee AI Key Project 17ZXRGGX00150.

REFERENCES

[1] http://www.jedit.org/.
[2] https://github.com/YangLin-George/TECCD.
[3] http://www.antlr.org/tools.html.
[4] http://www.thushv.com/natural_language_processing/word2vec-part-1-nlp-with-deep-learning-with-tensorflow-skip-gram/.
[5] https://www.oracle.com/technetwork/java/javase/downloads/index.html.
[6] https://ant.apache.org/.
[7] https://commons.apache.org/proper/commons-lang/.
[8] https://maven.apache.org/.
[9] https://opennlp.apache.org/.
[10] https://github.com/clonebench/BigCloneBench.
[11] https://deeplearning4j.org/neuralnet-overview.
[12] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. 2018.
[13] Ryota Ami and Hirohide Haga. Code clone detection method based on the combination of tree-based and token-based methods. *Journal of Software Engineering & Applications*, 10(13):891–906, 2017.
[14] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A simple but tough-to-beat baseline for s. 2016.
[15] Hamid Abdul Basit and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 513–516. ACM, 2007.
[16] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.
[17] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9), 2007.
[18] Lutz Büch and Artur Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104. IEEE, 2019.
[19] Per Erik Danielsson. Euclidean distance mapping. *Computer Graphics Image Processing*, 14(3):227–248, 1980.
[20] Mayur Datar, Piotr Indyk, Nicole Immorlica, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Twentieth Symposium on Computational Geometry*, 2004.
[21] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE, 1999.
[22] Jianglang Feng, Baojiang Cui, and Kunfeng Xia. A code comparison algorithm based on ast for plagiarism detection. In *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on*, pages 393–397. IEEE, 2013.
[23] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330. ACM, 2008.
[24] David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. In *ACM SIGCSE Bulletin*, volume 31, pages 266–270. ACM, 1999.
[25] Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *IEEE International Conference on Neural Networks*, pages 347–352 vol.1, 2002.
[26] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.
[27] G. E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9(2):2579–2605, 2008.
[28] Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Annual report of Osaka University: academic achievement*, 2001:22–25, 2002.
[29] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.
[30] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering*, pages 171–183, 1993.
[31] J Howard Johnson. Substring matching for clone detection and change tracking. In *ICSM*, volume 94, pages 120–126, 1994.
[32] Cory J Kapser and Michael W Godfrey. cloning considered harmful considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645, 2008.
[33] Egambaram Kodhai and Selvadurai Kanmani. Method-level code clone detection through lwh (light weight hybrid) approach. *Journal of Software Engineering Research & Development*, 2(1):12, 2014.
[34] Kostas Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 44–54. IEEE, 1997.
[35] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2):77–108, 1996.
[36] Rainer Koschke. Incremental clone detection. In *European Conference on Software Maintenance and Reengineering*, pages 219–228, 2009.
[37] Jens Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309. IEEE, 2001.
[38] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436, 2015.
[39] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
[40] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 249–260. IEEE, 2017.
[41] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.
[42] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881. ACM, 2006.
[43] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 106–115. IEEE, 2007.
[44] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *icsm*, volume 96, page 244, 1996.
[45] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
[46] Tomas Mikolov, Martin Karafit, Lukas Burget, Jan Cernock, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH 2010, Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September*, pages 1045–1048, 2010.
[47] Narcisa Andreea Milea, Lingxiao Jiang, and Siau-Cheng Khoo. Scalable detection of missed cross-function refactorings. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 138–148. ACM, 2014.
[48] Narcisa Andreea Milea, Lingxiao Jiang, and Siau-Cheng Khoo. Vector abstraction and concretization for scalable detection of refactorings. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 86–97. ACM, 2014.
[49] Manishankar Mondal, Md Saidur Rahman, Ripon K Saha, Chanchal K Roy, Jens Krinke, and Kevin A Schneider. An empirical study of the impacts of clones in software maintenance. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 242–245. IEEE, 2011.
[50] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 87–94. IEEE, 2002.
[51] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Gapped code clone detection with lightweight source

code analysis. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 93–102. IEEE, 2013.

[52] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, Yang Liu, and Santhoshkumar Saminathan. subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs. 2016.

[53] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[54] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[55] Matthias Rieger. *Effective clone detection without language barriers*. PhD thesis, Verlag nicht ermittelbar, 2005.

[56] Chanchal K Roy and James R Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 172–181. IEEE, 2008.

[57] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[58] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 18–33. IEEE, 2014.

[59] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541(115):64–68, 2007.

[60] Tobias Sager. Detecting similar java classes using tree algorithms. In *International Workshop on Mining Software Repositories*, pages 65–71, 2006.

[61] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc:scaling code clone detection to big-code. In *Ieee/acm International Conference on Software Engineering*, pages 1157–1168, 2016.

[62] Abdullah Sheneamer, Swarup Roy, and Jugal Kalita. A detection framework for semantic code clones and obfuscated code. *Expert Systems with Applications*, 97, 2017.

[63] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *IEEE International Conference on Software Maintenance Evolution*, 2014.

[64] Jeffrey Svajlenko and Chanchal K Roy. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 596–600. IEEE, 2016.

[65] Vera Wahler, Dietmar Seipel, J Wolff, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 128–135. IEEE, 2004.

[66] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. Ccaligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1066–1077. ACM, 2018.

[67] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 3034–3040, 2017.

[68] Richard Wettel and Radu Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, page 8 pp., 2006.

[69] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[70] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. [acm press the 2017 acm sigsac conference - dallas, texas, usa (2017.10.30-2017.11.03)] proceedings of the 2017 acm sigsac conference on computer and communications security - ccs î7 - neural network-based graph embedding for cross-platform binary code. 2017.

[71] Wuu Yang. Identifying syntactic differences between two programs. *Software Practice & Experience*, 21(7):739–755, 1991.

[72] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *Eprint Arxiv*, 2014.