Functional Code Clone Detection with Syntax and Semantics Fusion Learning

Chunrong Fang[*]
State Key Laboratory for Novel Software Technology
Nanjing University
China
fangchunrong@nju.edu.cn

Zixi Liu
State Key Laboratory for Novel Software Technology
Nanjing University
China
lzxcici@qq.com

Yangyang Shi
State Key Laboratory for Novel Software Technology
Nanjing University
China
doublesea.shi@qq.com

Jeff Huang
Parasol Laboratory
Texas A&M University
USA
jeffhuang@tamu.edu

Qingkai Shi
The Hong Kong University of Science and Technology
China
qshiaa@cse.ust.hk

ABSTRACT

Clone detection of source code is among the most fundamental software engineering techniques. Despite intensive research in the past decade, existing techniques are still unsatisfactory in detecting "functional" code clones. In particular, existing techniques cannot efficiently extract syntax and semantics information from source code. In this paper, we propose a novel joint code representation that applies fusion embedding techniques to learn hidden syntactic and semantic features of source codes. Besides, we introduce a new granularity for functional code clone detection. Our approach regards the connected methods with caller-callee relationships as a functionality and the method without any caller-callee relationship with other methods represents a single functionality. Then we train a supervised deep learning model to detect functional code clones. We conduct evaluations on a large dataset of C++ programs and the experimental results show that fusion learning can significantly outperform the state-of-the-art techniques in detecting functional code clones.

CCS CONCEPTS

• **Software and its engineering** → *Functionality*; *Maintaining software*;

KEYWORDS

Code clone detection, functional clone detection, code representation, syntax and semantics fusion learning

[*]Corresponding author

ACM Reference Format:
Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Los Angeles, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397362

1 INTRODUCTION

Code clone detection is fundamental in many software engineering tasks, such as refactoring [2, 4, 20, 33, 34], code searching [1, 24, 26], reusing [6, 12], and bug detection [10, 17]. If a defect is identified in a code snippet, all the cloned code snippets need to be checked for the same defect. As a result, code clones can lead to error propagation, which can severely affect maintenance costs. Thus, code clone detection is substantial in software engineering, and has been widely studied.

Code clone can be divided into four types according to different similarity levels [3]. **Type-1** (Exactly same code snippets): two identical code snippets except for spaces, blanks, and comments. **Type-2** (Rename/parameterize code): same code snippets except for the variable name, type name, literal name, and function name. **Type-3** (Almost identical code snippets): two similar code snippets except for several statements added or deleted. **Type-4** (Functional clones): heterogeneous code snippets that share the same functionality but have different code structures or syntax. Type-1, Type-2, and Type-3 code clones are well detected by many existing approaches [28]. However, there are still some unresolved issues on detecting Type-4 code clones [28]. The Type-4 code clone detection is the most complicated process because the syntax and semantics of source code are flexible. Detecting functional code clones is challenging because the code representations not only need to represent the syntax of source code, but also need to reflect the structure and relationship between code snippets. In order to detect the functional code clones effectively, many approaches try to use syntax-based or semantics-based information to represent source code.

One of the commonly-used syntax-based code representations is AST(Abstract Syntax Tree), which can represent the syntax of each statement. In some cases, code snippets with different syntax can implement the same functionality, which can be regarded as functional clones. For example, the statement "i = i + 1" and "i += 1" are not identical if we calculate the text-similarity directly. However, these two statements share the same AST. Typical approaches [9, 27] use a syntax parser to parse source code into a syntax tree.

Another commonly-used semantics-based code representation is PDG (Program Dependence Graph). PDG is a graph notation that contains both data dependence and control dependence. PDG can divide code snippets into basic code blocks and characterize their structures. The same functional code snippets with different structures can be detected as clones. For example, the loop structure *for* and *while* have different AST but their PDGs are identical. Typical approaches include Duplix [15] and GPLAG [18] which detect clones by comparing the isomorphic graphs of code pairs.

However, both syntax-based and semantics-based approaches have their limitations. On the one hand, although AST can represent the abstract syntax of source code, it is not able to capture the control flow between statements. On the other hand, each node in PDG is a basic code block (i.e., a statement at least), which is too coarse [36] compared with AST. The improper granularity may lead to a lack of details inside code blocks. For example, for the statement "int a = b + c", the AST nodes are *int, =, +, a, b, c.* However, this overall statement is stated in a node of PDG. Besides, the PDG for complex codes can be extremely complicated, and computing graph isomorphism is not scalable [18].

Moreover, AST is commonly used to represent a method, while PDG is used to represent an overall file. Thus, most existing approaches compare code similarity at the granularity of a single method or a single file [30, 38]. A functionality may consist of multiple methods. In other words, a single method may not express the complete implication of a functionality. Meanwhile, a file may contain several unrelated functionalities. Consequently, these granularity levels may not be the best choice for functional clone detection.

To overcome the limitations mentioned above, we use the call graph to combine related methods, so the granularity is flexible, and it depends on the granularity of real functionality. Additionally, we use CFG (Control Flow Graph) to represent the code structure instead of the more complicated PDG. CFG can capture the same control dependence as PDG, and it can overcome the time-consuming obstacle. Each method can be generated into an AST and CFG. As the representations of AST and CFG have different structures, they cannot be fused directly. Therefore, we apply embedding learning techniques [7] to generate fixed-length continuous-valued vectors. These vectors are linearly structured, and thus the syntactic and semantic information can be fused effectively. Note that these vectors are particularly suitable for deep learning models.

In this paper, we propose a novel approach to detect functional code clones through syntax and semantics fusion learning. Particularly, we analyze the method call relationship before extracting the syntactic and semantic features. Our basic idea is to identify the similar functionality of different code snippets by analyzing the call graph and combine the syntactic and semantic features based on AST and CFG by embedding techniques [7]. We use AST and CFG to represent the syntactic and semantic features, respectively. With such features, we further train a DNN model to detect functional code clones. We evaluated our approach on a large functional clone dataset OJClone [22] for C/C++ programs, which contains 104 programming tasks, and each task has 500 source codes submitted by different students. The different source code files for the same task can be regarded as functional clones. The experimental results show that our approach outperforms the state-of-art techniques with F1 value 0.96, including DECKARD [9], SourcererCC [30], CDLH [38], DeepSim [42], and ASTNN [41].

In summary, the main contributions of this paper are as follows.

(1) We propose a fine-grained granularity of source code for functionality identification. We regard the methods with caller-callee relationships as the implementation of a functionality. To the best of our knowledge, our approach first consider call graph in functional code clone detection.

(2) We propose a novel code joint representation with embedding learning of both syntactic and semantic features. With the combination of syntactic and semantic information, functional code clones can be detected precisely.

(3) We present the design and implementation of the proposed approach for C++ programs. Our extensive evaluation on a large real-world dataset shows a promising result on functional code clone detection. We release all the data used in our studies.[1] In particular, we include all the source codes, datasets, code representations, embedding features, and analysis results.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivation of our approach. Section 3 illustrates our approach in detail. Section 4 shows our experimental evaluation on a large dataset. Section 5 represents the related work of clone detection. Finally, Section 6 concludes this paper.

2 BACKGROUND AND MOTIVATION

A typical code clone detection approach usually follows three steps. **(1) Code pre-processing:** remove irrelevant content in source codes, such as comments; **(2) code representation:** extract different kinds of abstractions in source codes, such as AST (Abstract Syntax Tree) [40] and PDG (Program Dependence Graph) [15]; **(3) code similarity comparison:** calculate the distance between two source codes. Code clone is detected when this distance reaches a threshold.

Since the code representation can have a crucial efftect on functional code clone detection, we mainly introduce the background of syntactic representation AST (Section 2.1) and semantic representation CFG (Section 2.2). These tree and graph structures cannot be used for networks directly, and we briefly discuss the background of word embedding (Section 2.3) and graph embedding (Section 2.4). The embedding techniques can be used to transform these tree and graph structures into feature vectors.

2.1 Abstract Syntax Tree

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. [2] This tree defines the structure of source codes. By manipulating this tree, researchers can precisely locate declaration statements, assignment statements, operation statements, etc., and

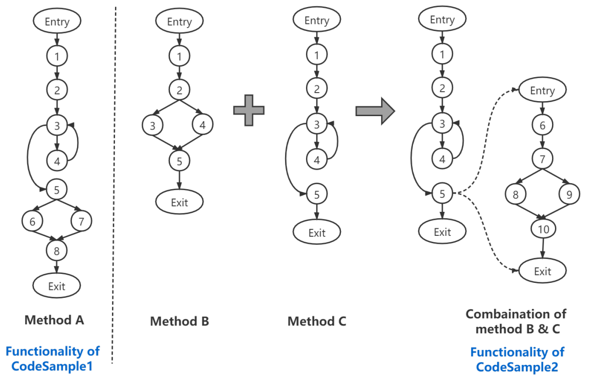
[1]It is publicly available at: https://github.com/shiyy123/FCDetector
[2]https://en.wikipedia.org/wiki/Abstract_syntax_tree

```
int A() {
    int res = 1, i = 0;
    int len = 10, flag = 100;
    for(; i < len; i++){
        res += i;
    }
    if (res > flag){
        res = res - flag;
    }
    else{
        res = flag - res;
    }
    return 0;
}
int D() {
    //do something
}
```

(a) CodeSample1

```
int B(int a, int b) {
    int c;
    if (a > b){
        c = a - b;
    }
    else{
        c = b - a;
    }
    return c;
}
int C() {
    int res = 1, i = 0;
    int len = 10, flag = 100;
    while(i < len){
        res += i;
        i ++;
    }
    res = B(res, flag);
    return 0;
}
```

(b) CodeSample2


Entry
Exit
Method A
Functionality of CodeSample1
Method B
Method C
Combaination of method B & C
Functionality of CodeSample2

(c) CFGs of method A, B, and C. The CFG combining method B and C is equivalent to that of A.

Figure 1: Using call graph to identify the functionality

realize operations such as analyzing, optimizing and changing the codes.

Some studies use AST in token-based approaches for source code clone detection [9], program translation [19], and automated program repair [39]. Due to the limitation of token-based approaches [28], these approaches can capture little syntactic information of source code.

2.2 Control Flow Graph

Control Flow Graph (CFG) is a representation, using graph notation, of which all paths that might be traversed through a program during its execution. [3] In each CFG, there is a basic block containing several code statements. CFG can easily encapsulate the information per each basic block. It can easily locate inaccessible codes of a program, and syntactic structures such as loops are easy to detect in a control flow graph.

2.3 Word Embedding

Word embedding is a collective term for language models and representation learning techniques in natural language processing (NLP). Conceptually, it refers to embedding a high-dimensional space with the number of all words into a continuous vector space with a much lower dimension, and each word or phrase is mapped into a vector on the real number field.

Since the source code and natural language are similar in some ways, many approaches try to use word embedding techniques to process the source code. In our approach, we first traverse each node of AST in preorder and then use word embedding techniques to transform AST into vectors.

2.4 Graph Embedding

Graphs, such as social networks, flow graphs, and communication networks, exist widely in various real-world applications. Real graphs are often high-dimensional and difficult to process. Researchers have designed graph embedding algorithms as part of the dimension reduction technique. Most existing graph embedding techniques aim to transform each node of the graph into a vector. Graph algorithms can be used in different graph tasks, such as graph classification, link prediction, and graph visualization.

Since the CFG is a typical graph, some existing approaches try to use graph embedding techniques for code representation. For example, Tufano et al. [35] uses a kind of graph embedding techniques to represent the semantic features. In our approach, we compare several graph embedding techniques and choose the most effective technique for CFG representation.

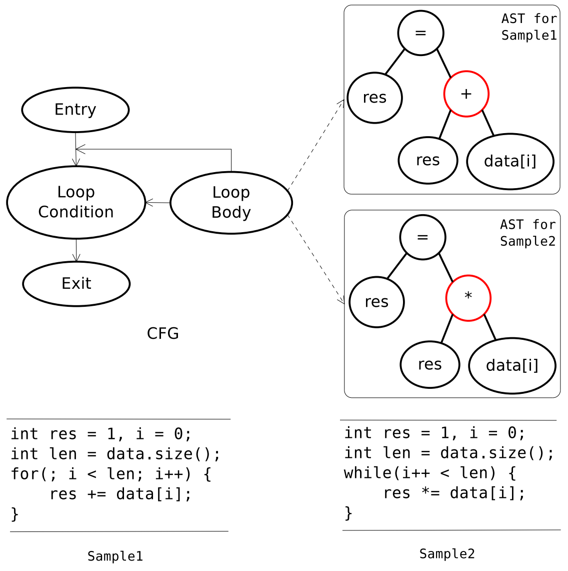
2.5 Motivating Example

To detect functional code clones, we first extract the call graph and analyze the functionality of each file. We regard the methods with caller-callee relationships as a functionality. The method without any caller-callee relationship with other methods can be regarded as a self-contained functionality.

For the two source code files shown in Figure 1(a) and 1(b), a method A in $CodeSample1$ implements a simple calculation functionality. In $CodeSample2$, a method C calls a method B, and their combination implements the same functionality as the method A. As shown in Figure 1(c), if we detect code clones in method granularity, we can see that the CFGs of method A, B, and C are structurally different. However, after we combine method B and C according to the call graph, we can find the control flow of the combination is equivalent to method A. Therefore, after considering the call graph between methods, we can conclude that method A in $CodeSample1$ and the combination of method B and C in $CodeSample2$ are functional clones. If we only detect the functionality at method level, the result will be misleading. Besides, it is also inappropriate to detect functional clones in file granularity. Please note that there is a method D in $CodeSample1$, which is irrelevant to the method A, and it is unreasonable to put them together directly. Thus, it is more suitable to extract the call graph and identify the functionality before capturing code features.

After determining the functionalities of each file, we detect functional code clones by analyzing the AST and CFG. CFG can reflect

[3] https://en.wikipedia.org/wiki/Control-flow_graph



```
int res = 1, i = 0;
int len = data.size();
for(; i < len; i++) {
    res += data[i];
}
```

```
int res = 1, i = 0;
int len = data.size();
while(i++ < len) {
    res *= data[i];
}
```

Figure 2: AST and CFG for Sample1 and Sample2

the structure of statements and show the control flow, which is shown in Figure 1(c). The control flow is abstract, and we cannot clearly identify the differences between nodes in CFG. Thus, we detect code clones with syntax and semantics fusion learning. We can capture the syntax by analyzing AST. For instance, if the symbol "+=" in the *for* loop is changed into "*=" in method *A*, the functionality is entirely changed. However, since the CFG structure is not changed, it will lead to a false positive result if an approach only takes CFG into consideration. The difference between these two code snippets lies inside the AST of two loop bodies, as shown in Figure 2. Therefore, it is of necessity to consider both syntax and semantics for functional code clone detection.

Moreover, some kinds of code structures can achieve the same functionality, but the AST structures are completely different. For example, both of the *for loop* and *recursive* structure can calculate the factorial of a given input. The ASTs of these two code structures are completely different. However, the inter-procedural control-flow graphs of the two code snippets are very similar, both of which contain a loop. Since our approach not only analyzes the AST, but also utilizes inter-procedural information (such as the call graph), our approach is effective at detecting such inter-procedural code clones.

3 APPROACH

As illustrated in Figure 3, our proposed approach consists of the following three components.

(1)Identify the functionality with call graph. To identify the functionality in each source code file, we extract the call graph to analyze the caller-callee relationships among methods. The methods with caller-callee relationships are combined together as a functionality. The method without any caller-callee relationship with other methods is regarded as a self-contained functionality.

(2) Extract syntactic and semantic representations. In order to capture the code features, we need to extract the AST and CFG to represent syntax and semantics. First, we extract the AST and CFG of each method. Then, we combine the AST and CFG of related methods according to the call graph. The combined AST/CFG represents a separate functionality. Finally, we use embedding techniques to encode AST and CFG into syntactic and semantic feature vectors.

(3) Train a DNN model. To transform the functional clone detection into binary classification, we train a DNN(Deep Neural Network) classifier model with labeled data. When encountered two new functions, we extract and fuse feature vectors and use the trained model to predict whether they are code clones or not.

3.1 Identifying the Functionality with Call Graph

We extract the call graph to identify the functionality in each source code file. The call graph represents calling relationships between methods in a program. Each node represents a method, and each edge (f, g) indicates that the method f calls method g. For the input source code files, all statements in them are marked with a globally unique id, which can be regarded as an identifier. The caller-callee relationship is expressed in the form of a triple

$\langle callerId, statementId, calleeId \rangle$.

$callerId$ and $calleeId$ are the statement id of the corresponding method, and $statementId$ is the id of call statement. For example, in Figure 1(b), the method *C* calls the method *B*. Therefore, the caller statement is *int C()* in line 10, the call statement is *c = B(a, b, flag);* in line 14, and the callee statement is *int B(int a, int b, bool flag)* in line 1. According to the call graph expression, we can obtain the connected methods as a functionality.

To illustrate the caller-callee relationships, we list six call graph cases, as shown in Figure 4. These six cases of static call graph are the most commonly used call graph. Our approach includes but is not limited to these six caller-callee relationships. The caller-callee relationships of methods are explained below. In case (1), there is a functionality that includes method *A*; in case (2), method *A* has a recursive call for itself, and there is also a functionality that includes a single method *A*; in case (3), there exists method *A* calls method *B*, and this case has a functionality that includes method *A* and *B*; in case (4), method *A* and method *B* call each other, and there is a functionality that includes method *A* and *B*; in case (5), method *A* calls method *B* and method *A* calls method *C*, but there is also only one functionality, and the functionality includes method *A*, *B*, and *C*; in case (6), method *B* calls method *A* and method *C* calls method *A*. There are two functionalities, one including methods $\{A, B\}$ and another including methods $\{A, C\}$.

3.2 Extracting Syntactic and Semantic Representations

3.2.1 AST representation. Suppose that C is a code snippet C and N_{root} is its corresponding AST entry node. To extract the syntactic representation of C, we start from N_{root} and iterate through all the nodes of AST in preorder. In each AST node, there is an identifier, such as the symbols and variable names. The identifier sequence

$$Seq = \{ident_1, ident_2, \ldots, ident_n\}$$

generated in this process can be used to represent the syntactic information of C.

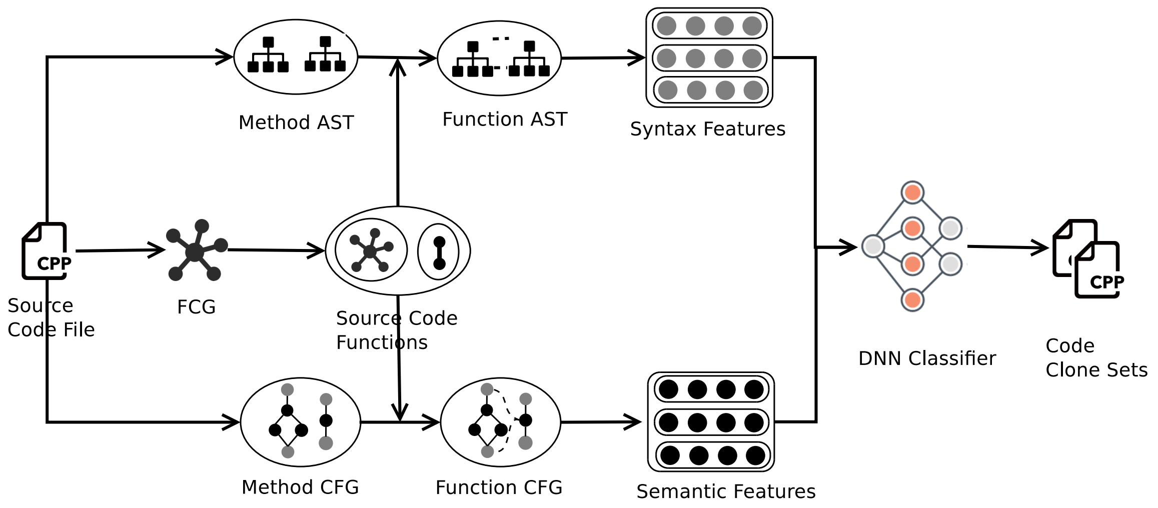


Figure 3: The architecture of our approach

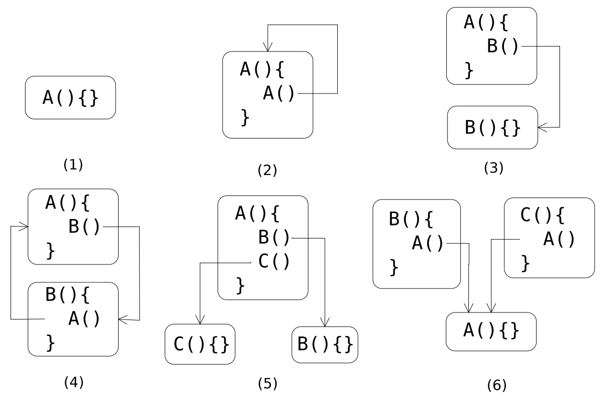


Figure 4: Six kinds of caller-callee relationships among methods

The naming of variables may vary greatly in different programs. Thus variable names can have an effect on functional clone detection. Given an identifier sequence *Seq*, we first normalize it by replacing constant values and variables by its type: $\langle int \rangle$, $\langle double \rangle$, $\langle char \rangle$ and $\langle string \rangle$, and add a global self-increasing number, like $\langle int_1 \rangle$. When encountered a decimal, no matter it is a $\langle float \rangle$ or $\langle double \rangle$, we unify them as $\langle double \rangle$.

We first extract the AST of each method and obtain the corresponding identifier sequence *Seq*. Then according to the caller-callee relationships extracted by call graph stated in Section 3.1, we put the AST of each connected method together as a set of AST. Finally, for the methods with connections, we use the set of their AST to represent the syntax of the functionality. Similarly, for the method without caller-callee relationship, we use the corresponding AST to represent the syntax of the functionality.

3.2.2 CFG representation. Compared to AST, CFG captures more comprehensive semantic information such as branch and loop. Although CFG also contains syntactic information at the level of basic blocks, its representation is too coarse for detecting functional clones compared to the syntactic information extracted from AST at identifier-level granularity. Thus we focus on extracting semantic information from CFG.

Given a code snippet C, we extract its CFG $G = (V, E)$ for each method, where V is a set of vertexes and E is a set of edges. Each vertex contains a statement of source codes, and the edges indicate the control flow of statements.

We first extract the CFG of each method and get the corresponding graph G. Then according to the call graph stated in Section 3.1, we connect the CFGs and generate a larger CFG to represent a functionality. The specific connection rules are presented as follows. In case (1), the CFG of a functionality is identical to the CFG of the method; in case (2), the CFG of a functionality is also identical to the CFG of the method, and an edge is added from caller statement to the entry node of the method; in case (3), first we need to add the CFG of method A and B, then we obtain the parent nodes list and the child nodes list of the functionality call statement in method A, and then all the nodes are pointed to the entry node in method B, and all the child nodes point to the exit node in method B; case (4), (5) are similar to case (3), and the edges related to another function call statement are modified, additionally; in case (6), there are two functionalities, so we need to process these two functionalities respectively.

3.2.3 Embedding syntactic feature of functionality. Each method in source code is now represented as a sequence of identifiers. We then use Word2vec [21], a word embedding technique, to encode syntactic features. Word embedding is a general term of language model and embeds high-dimensional space words into a much lower dimensional continuous vector space, where each word is mapped to a vector in the real number field.

As shown in Figure 5, in order to encode the syntactic features of each method, the normalized identifier sequence *Seq* for each method are put together as the corpus. Word2vec uses the corpus as the input of training model, and output a set of fixed-length vectors to represent each AST node. To get a syntactic feature vector of each

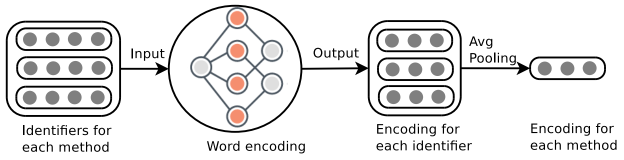


Figure 5: Syntactic feature embedding

method, we applies average pooling to all the vectors of identifiers in each method. After this step, the syntactic information of each method is represented as a fixed-length feature vector

$$mv = average(h_i), i = 1, \ldots, N$$

Where h_i is the feature vector of each identifier.

For each functionality (i.e., the connected methods), we then apply average pooling on all the methods to produce the corresponding syntactic feature. The syntactic information of each functionality is represented as a fixed-length feature vector

$$fv = average(mv_i), i = 1, \ldots, N$$

Where mv_i is the syntactic feature vector of each method.

3.2.4 Embedding semantic feature of functionality. We use graph embedding techniques to encode the CFG. Graph embedding is to use a low dimensional and dense vector to represent every node in the graph. The vector representation can reflect the structure in the graph. In essence, the more adjacent nodes the two nodes share (i.e., the more similar the contexts of the two nodes are), the closer the corresponding vectors of two nodes are. After graph embedding, each method is represented as a fixed-length feature vector. There are many graph embedding techniques, e.g. Graph2vec [23] , HOPE [25], SDNE [37], and Node2vec [5].

Different from other graph embedding techniques, Graph2vec can generate a vector to reflect the feature of an overall graph. However, other techniques mentioned above can only generate a vector for each node in the graph, and the feature vector of the overall graph is the average value of all nodes. Thus, we use Graph2vec to generate feature vectors instead of the other techniques. We compare different embedding techniques and the results in Table 4 show that Graph2vec performs the best F1 value.

For each method, we use the feature vector generated by Graph2vec to represent the semantic feature vector. For each functionality, we connect all the CFGs of connected methods according to the call graph, which is illustrated in Section 3.1. According to the connection rules stated in Section 3.2.2, we can obtain the connected CFG of connected methods. Then we apply graph embedding on the connected CFG of this functionality. Finally, we transform the connected CFG into feature vectors using the technique of graph embedding.

3.3 Training a DNN Model

To detect functional code clones, we may directly compare the Euclidean distance between the joint feature vectors of syntactic and semantic encodings at functionality granularity, as described in the previous section. This distance-based approach has been used in prior work to detect syntactic clones, such as Deckard [9]. However, calculating the distance directly does not work in our case because the distance between the extracted joint feature vectors is irregular, as illustrated in Figure 6. This is because the weight of each dimension is different, and it is difficult to manually set an appropriate threshold to distinguish between functional code clones.

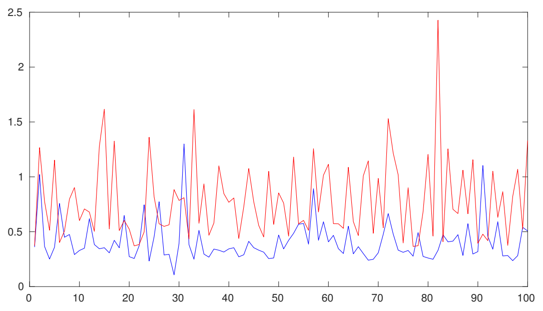


Figure 6: Euclidean distance between feature vectors. The blue line represents the distance between similar code snippets, and the red line represents the distance between dissimilar code snippets. The horizontal axis is the id of code snippets

To address this problem, we propose to use a deep learning model that can effectively detect functional clones between code snippets. In particular, we fuse syntactic and semantic feature vectors as input to train a feed-forward neural network [32], and convert code clone detection to binary classification with the softmax layer at the end of the model. Instead of choosing a specific distance metric or set the threshold for clone detection, we adopt deep learning techniques for functional code clone detection that can automatically learn latent code features from the fused feature vectors.

The architecture of the DNN model is shown in Figure 7, which consists of four components. Firstly, we obtain the syntactic and semantic features following Section 3.2.4 and concatenate the two features together as the input of the model, where syntactic and semantic information are both represented with a 16-dimensional vector.

To obtain word embeddings and graph embeddings for AST and CFG, we use Word2vec and Graph2vec to generate word embeddings of length {4, 8, 16, 32}, and graph embeddings of length {4, 8, 16, 32}. We choose the vector length as 16, according to the experiment. In short, the longer vector will increase the training time, and the shorter vector cannot capture sufficient code features. The detailed experiment is illustrated in Section 4.4.

As for the input of the DNN model, we use a pair of code features to represent whether the two functionalities are clones. The input vector contains three portions: the fusion feature vector V_1 for the first functionality, the fusion feature vector V_2 for the second functionality, and a label. The fusion feature vector contains a 16-dimensional syntactic vector and a 16-dimensional semantic vector. The label is a boolean value, where 0 represents non-clone pair, and 1 represents clone pair. Thus, the total dimensions of the input are 65(32+32+1).

The order of the fusion features of two functionalities, $[V_1, V_2]$ and $[V_2, V_1]$, may affect the classification results. Therefore, we

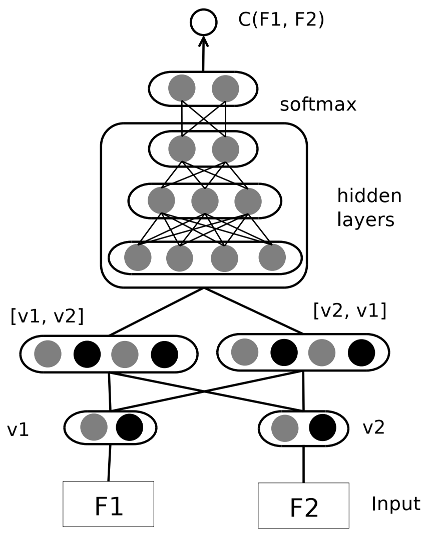


Figure 7: The architecture of the deep fusion learning model

place a fully connected layer before the input of the two features. Then, the hidden layer uses linear transformation followed by a squashing non-linearity to transform the inputs into the neural nodes in the binary classification layer. This step can provide a complex and non-linear hypothesis model, which has a weight matrix W to fit the training set. The feed-forward step has been completed so far. In the next component, this model then uses back-propagation to adjust the weight matrix W according to the training set. Finally, there is a softmax layer, which converts the functional clone detection into a classification task.

Suppose there are two code fusion vectors V_1 and V_2, and their distance is measured by $d = |v_1 - v_2|$ for syntactic and semantic relatedness. Then we treat the output as their similarity:

$$\hat{y} = softmax(\hat{x}) \in [0, 1]$$
$$\hat{x} = Wor + bo$$

Where Wor is the weight matrix of hidden layers. This model uses a cross-entropy loss function, which performs better than mean-squared loss function in updating weight. The goal of training the model is to minimize the loss. We choose AdamOptimizer [13] because it can control the learning rate in a certain range, and the parameters are relatively stable. The model is a DNN classifier and has two classes.

After all the parameters are optimized, the trained models are stored. For new code fragments, they should be preprocessed into syntactic and semantic vectors and then fed into the model for prediction. The output is 1 or 0, which represents clone or non-clone code pairs.

4 EXPERIMENTAL EVALUATION

With the implementation of our approach, we conducted evaluations to explore the following research questions.

- RQ1: How does our approach perform in functional code clone detection comparing to state-of-the-art approaches?
- RQ2: How does fused code representations perform in functional code clone detection?
- RQ3: How does our chosen embedding techniques perform in functional clone detection?
- RQ4: How does DNN perform in functional code clone detection comparing to other traditional machine techniques?

4.1 Experimental Settings

We used joern to abtain ASTS and CFGs of C++ source code. To extract syntactic features, we trained embeddings of tokens using word2vec with Skip-gram algorithm and set the embedding size to be 16. To extract semantic features, we trained embeddings of graph using Graph2vec and set the embedding size to be 16. There are 5 hidden layers of the DNN classifier, and the dimension of each hidden layers are 128, 256, 512, 256, 128, respectively.

We evaluated our approach on a real-world dataset: OJClone [22] from a pedagogical program online judge system, which mainly includes C/C++ source code. OJClone contains 104 programming tasks, and each task contains 500 source code files submitted by different students. For the same task, the code files submitted by different students are considered as functional code clones. And each code snippet pair can be expressed with (s_1, s_2, y), where s_1 and s_2 are two code snippets and y is the code clone label of them. The value of y is $\{0, 1\}$. If s_1 and s_2 are solving the same task, the value of y is 0. If s_1 and s_2 are solving different tasks, the value of y is 1. We choose the first 35 problems from the 104 problems with 100 source files from each problem.

Then, we generated 100*100 clone pairs for each task and 100*100 non-clone pairs for two different tasks. When the problem set is more than two, the non-clone pairs will be far more than clone pairs. The imbalance of data will affect the experimental results, so we adopt under-sample to balance the clone pairs and non-clone pairs. We randomly divide the dataset into two parts, of which the proportions are 80% and 20% for training and testing. We use AdamOptimizer [13] with learning rate 0.001 for training and the number of training epochs is 10000. All the experiments are conducted on a server with 8 cores of 2.4GHz CPU and a NVIDIA GeForce GTX 1080 GPU.

4.2 The Effectiveness of Our Approach

RQ1: How does our approach perform in functional code clone detection comparing to state-of-the-art approaches? To evaluate the performance of our approach, we compare our approach with several state-of-the-art code clone approaches as follows:

- Deckard, a syntax-tree-based approach, which uses Euclidean distance to compare the similarity between code snippets to detect code clone.
- SourcererCC, a token-based code clone detector for very large codebases and Internet-scale project repositories.
- The approach proposed by White et al. converts source code into the defined tree structure and uses a convolution neural network to learn unsupervised deep features. Later we will name it DLC for short.
- CDLH, a deep learning approach to learn syntactic features of code clone, which converts code clone detection into supervised learning of hash characteristics of source code.
- DeepSim, a semantic-based approach that applies supervised deep learning to measure functional code similarity.

- ASTNN, a state-of-the-art neural network based source code representation for source code classification and code clone detection.

We compare our approach with these code clone detection approaches in terms of Precision (P), Recall (R), and F1-measure (F1). Because our dataset mainly belongs to functional code clones, the detection performance can indicate the ability of these approaches to detect functional clones.

Table 1: Results on OJClone

Tools	P	R	F1
Deckard	0.99	0.05	0.10
SourcererCC	0.07	0.74	0.14
DLC	0.71	0.00	0.00
CDLH	0.47	0.73	0.57
DeepSim	0.70	0.83	0.76
ASTNN	0.98	0.92	0.95
Our approach	**0.97**	**0.95**	**0.96**

Table 1 shows the precision, recall, and F1 values of the six approaches. It can be observed that CDLH, DeepSim, ASTNN and our approach can outperform the other three approaches: Deckard, SourcererCC, and DLC in terms of F1 value. As most of the code clones in OJClone belong to functional clones, the results means that CDLH, DeepSim, ASTNN and our approach can well deal with functional clones, while the other three approaches cannot. CDLH, DeepSim and our approach leverage the supervised information to adjust the training process and improve the clone detection model, while the other three are unsupervised approaches. This result means unsupervised approaches can hardly learn the similarity between functional clones. For Deckard, the syntactic information of the source code is embedded into a feature vector, and we calculate the Euclidean distance between feature vectors to detect code clones. In the process of calculating Euclidean distance, the weight of each dimension is considered the same. However, for syntactic feature vectors in functional code clone detection, the importance of each dimension in the feature vector is tend to be different. So the recall of Deckard is pretty low, but the precision becomes very high when the syntactic information of the two code snippets happen to be identical. SourcererCC obtains high recall while getting pretty low precision since it mistakes too many code snippet pairs as code clones. This is because the variable name of these code snippets submitted by students are not standard, and students tend to use simple names like *a*, *b*, and *c*. SourcererCC mistakes these code snippets with the same tokens as code clone. DLC is similar to Deckard and is an approach using the syntactic features to train the convolutional neural network [11] also failed to detect functional code clones without the guidance of supervised information.

Compared with CDLH, a syntax-based deep learning clone detection approach, our approach achieves much higher precision and a little higher recall. Compared with DeepSim, a semantics-based deep learning clone detection approach, our approach achieves higher precision at the sacrifice of a little recall. Compared with ASTNN, an AST-based deep learning clone detection approach, our approach achieves higher recall at the sacrifice of a little precision. This is mainly because our approach leverages both syntactic and semantic information in functionality granularity. In addition, since our approach utilizes inter-procedural information (such as the call graph), we are good at detecting inter-procedural code clones, which is demonstrated in Section 2.5.

We evaluated the time performance of approaches mentioned above. We ran each tool to detect code clones in OJClone with the optimal parameters. And we run each tool three times and report the average.

Table 2: Time performance on OJClone

Tools	Training Time	Prediction Time
Deckard	-	32s
SourcererCC	-	30s
DLC	120s	67s
CDLH	8307s	82s
DeepSim	14545s	34s
ASTNN	9902s	72s
Our approach	**8043s**	**63s**

Table 2 reports the time performance of these tools. Deckard and SourcererCC do not need to train the model and conduct clone detection directly. Thus, we use "-" in training time for Deckard and SourcererCC. DLC, CDLH, DeepSim, ASTNN, and our approach need to train the clone detection model. Our approach also needs to conduct embedding learning for syntactic and semantic information. Thus it takes the long time in the training step. Although the training time of DeepSim, CDLH, ASTNN, and our approach are too much compared to the other three approaches, it is a one-time offline process. Once the model is trained, it can be reused to detect code clones. DTC incorporates a simple deep learning model, which cost much less time. Compared with other approach, especially ASTNN, which is the state-of-the-art approach, the training time of our approach is much shorter. Thus, our approach is more efficient. In the long run, the use of deep learning is more convenient, and the trained model can be reused to detect code clones.

4.3 The Effectiveness of Fused Code Representations

RQ2: How does fused code representations perform in functional code clone detection? In the previous section, we have validated the superiority of our approach on functional clone detection. In this section, we further validate the effectiveness of the syntax and semantics fusion learning.

For this purpose, three different code representations, including text, AST, and CFG, are used to compare the detection performance. Code tokens is also a common repsentation of source codes. However, AST is a higher abstraction of code tokens and contains more syntactic information than code tokens[4]. And we do not choose code tokens in our experiment. To extract the text sequence of

[4]https://stackoverflow.com/questions/25049751/constructing-an-abstract-syntax-tree-with-a-list-of-tokens/

source codes, we use the Turing eXtender Language proposed in NICAD [29], a famous text-based clone detection approach. The extraction of AST and CFG has been detailed in the previous section.

Given a source code snippet and three code representations $R = \{R_1, R_2, R_3\}$, where the three representations are text, AST and CFG in turn. Word2vec is used to conduct embedding learning for the first two representations and Graph2vec for CFG. Then the embeddings $E = \{E_1, E_2, E_3\}$ are for the three representations and each embedding is a fixed length continuous-valued vectors. For each fusion representation FR, a tuple $t = h_1, h_2, h_3$ is generated, where $h_i = True$ means FR contains this representation and $h_i = False$ otherwise. t can assume 8 possible values, i.e., $t = \{FFF, FFT, \ldots, TTT\}$. Among them, FFF means that none of the three representations are selected, which make no sense for answering RQ1, and thus FFF is eliminated. These FR sets are taken as input to and evaluate the precision, recall and F1-measure.

Table 3: Precision (P), recall (R) and F1 of all representations

ID	Text	AST	CFG	P	R	F1
1	F	F	T	0.82	0.92	0.87
2	F	T	F	0.91	0.62	0.74
3	**F**	**T**	**T**	0.97	**0.95**	**0.96**
4	T	F	F	**0.99**	0.21	0.35
5	T	F	T	0.85	0.52	0.65
6	T	T	F	0.94	0.44	0.60
7	T	T	T	0.98	0.92	0.95

Table 3 shows the precision, recall, and F1-measure of seven combinations of code representations. It can be observed that $ID = 3(FTT)$, the fusion of AST and CFG, achieves the highest F1 values than others, and it validates that our proposed code representation can capture the semantics of the source code for functional clone detection. For $ID = 1(FFT)$, the recall of this representation is pretty high. It is because the source codes in the dataset have relatively simple control flow, and many functionality pairs belonging to different problems are mistaken for code clones. For $ID = 2(FTF)$, the precision of it is high but has a relatively low recall. It is due to different students may realize the same functionality with different control flow, and this fusion cannot recognize this difference. Similarly, it can be observed that the same phenomenon as $ID = 4(TFF)$, which also has high precision and low recall. However, the recall of it is much lower than $ID = 2(FTF)$ because it is not appropriate to regard the source code as a natural language without any processing. Since the source codes submitted by these students tend to use the similar variable names, such as a, b and c, the similarity between the text is pretty high, which may lead to many mistakes for code clone. For $ID = 5(TFT)$ and $6(TTF)$, after adding the text representation, their recall have improved to a certain extent compared to $ID = 1(FFT)$ and $2(FTF)$, but precision are sacrificed a lot for the same reason as $ID = 4(TFF)$. In $ID = 7(TTT)$, we use all three representations. Intuitively, it is believed that the more kinds of code representations are used, the more syntactic and semantic information can be extracted, and the higher precision and recall is. However, the experiment result shows that too much information and too strict clone detection requirements greatly reduce the recall, and improve a little bit of precision. In summary, we choose to use a fusion of AST and CFG representations to conduct clone detection.

4.4 The Effectiveness of Word Embedding and Graph Embedding Techniques

RQ3: How does our chosen embedding techniques perform in functional clone detection? In the previous section, we have validated the effectiveness of our proposed code representations to conduct functional code clones. As we all know, there are several famous word embedding techniques: Word2vec and GloVe and graph embedding techniques: Graph2vec, HOPE, SDNE, and Node2vec. In this section, we shall validate the effectiveness of the word embedding and graph embedding techniques we choose for the embedding learning of the fusion of AST and CFG. For the purpose, the identifier sequence extracted from AST and the structure of CFG is taken as the input of these word embedding and graph embedding techniques to get the embeddings of them. Then we take the fusion of the two embeddings as the input of our deep fusion learning model and evaluate the precision, recall, and F1 measure under OJClone.

Table 4: Precision, recall and F1 of different embedding techniques

Embedding techniques	P	R	F1
Word2vec+Graph2vec	**0.97**	**0.95**	**0.96**
Word2vec+HOPE	0.88	0.79	0.84
Word2vec+SDNE	0.75	0.56	0.64
Word2vec+Node2vec	0.55	0.79	0.65
GloVe+Graph2vec	0.89	0.81	0.85
GloVe+HOPE	0.84	0.73	0.78
GloVe+SDNE	0.73	0.52	0.61
GloVe+Node2vec	0.54	0.87	0.67

Table 4 shows the precision, recall, and F1 measure of the combinations of two word embedding techniques and three graph embedding techniques. As shown in Table 4, for two kinds of word embedding techniques, Word2vec achieves better performance than GloVe. Compared with Word2vec, Glove adopted some overall statistics to make up for the shortcomings of the co-occurrence model, while the effect was not necessarily better than simple Word2vec in practice [16]. For the first three combinations, the combination of Word2vec and Graph2vec achieves the highest F1 value. That is because different from other approaches, the vectors generated by Graph2vec can reflect the overall graphs. The vectors generated by other approaches can only reflect the node structure, and the average value of all the nodes may lack the structural feature of the graph.

Sensitivity to the length of embeddings. In previous experimental settings, we use continuous-valued vectors of length 16 for word embeddings and graph embeddings. In this section, we study the influence of the different lengths of embeddings on functional clone detection performance of our approach, measured by F1 values. Table 5 shows the F1 value tends to be stable at around 0.96 after the length of word embeddings and graph embeddings are more than 16. Thus, we choose 16 as the length of the word embeddings and graph embeddings used in our approaches.

Table 5: The F1 values for different length of embeddings

word \ graph	8	16	32	64
8	0.63	0.71	0.76	0.76
16	0.79	**0.96**	0.93	0.95
32	0.81	0.95	0.96	0.94
64	0.83	0.96	0.96	0.95

4.5 The Effectiveness of Machine Learning Techniques

RQ4: How does DNN perform in functional code clone detection comparing to other traditional machine techniques? Table 6 reports the performance of SVM, logistic regression, and our approach on OJ-Clone. Compared with the traditional machine learning techniques, our approach can better capture the hidden features from syntactic and semantic vectors. Thus, our approach is efficient for functional code clone detection.

Table 6: Comparison with machine learning techniques

Techniques	P	R	F1
SVM	0.62	0.81	0.70
logistic regression	0.66	0.71	0.68
Our approach	**0.97**	**0.95**	**0.96**

4.6 Threats to Validity

There are two main threats to the validity. First, only OJClone dataset is used to demonstrate the effectiveness of our approach. However, OJClone is widely used to evaluate code clone detection approaches, such as Deepsim and ASTNN. We also try to turn to the BigCloneBench dataset, which is a widely-used benchmark for code clone detection. However, it is not proper for evaluating our approach, because the dataset does not contain inter-procedural programs while our approach aims to use caller-callee relation to detect inter-procedural code clones. Besides, our current experiment is on C++ programs, and we plan to involve other language in the future.

Second, as incorporating deep learning, the effectiveness of our approach is limited by the quality of the training set. Even we generated tens of thousands clone pairs and tried to deal with data imbalance. As we have release all the data in our studies, we also plan to build a large and representative dataset.

5 RELATED WORK

In code clone detection, the representation of source code determines the upper limit of source code information extraction, and it will determine the model design and affects the final performance.

According to different aspects of the use of source code, the representation method can be divided into four categories: text-based, token-based, syntax-based, and semantics-based techniques. Text-based techniques treat source code as text encoding, and token-based techniques parse sources code into tokens, which are then organized into token sequences. These two code representations are usually used to detect Type-1 and Type-2 code clones. Since our proposed approach is mainly dealing with functional code clone detection, which is regarded as Type-4 code clones, we do not focus on text-based and token-based techniques.

Syntax-based approaches are not sensitive to the order of source code and can detect Type-1, Type-2, Type-3, and partial Type-4 code clones. Syntax-based techniques take syntax rules of the source code into consideration and mainly contain two types: tree-based and indicators-based approaches. Deckard [9] uses syntax parser to parse source code into a syntax tree, then embeds syntax tree into features, finally uses a locally sensitive hash algorithm for clustering to find the similar code. Daniel Perez et al. [27] generates the AST of source code and uses a feed-forward neural network to classify to code clones. Different from these syntax-based approaches, we visit each node in AST by preorder and use word embedding techniques to learn the AST features.

Semantics-based techniques consider the semantic information of source code. Semantic information refers to the information that can reflect the functionality of code snippets. The most commonly used semantic representation is PDG, which is the combination of control dependence and data dependence. The approach proposed by Komondoor [14] uses program slices to find isomorphic subgraphs of PDGs. Sheneamer et al. [31] extract features from abstract syntax trees and program dependency graphs to represent a pair of code fragments to detect functional code clones. However, the shortcoming of PDG-based approaches is high time cost due to the complexity of PDG. Since the operational semantics of CFG is equivalent to that of deterministic PDG [8], we simply use CFG to represent the semantic features.

In addition to the four categories mentioned above, there are also some hybrid approaches for the functional code clone detection. For instance, CDLH [38] learns hash codes by exploiting the lexical and syntactic information for fast computation of functional similarity between code fragments. DeepSim [42], one of the state-of-the-art approaches, encodes both code control flow and data flow into a semantic matrix and uses a deep learning method to detect functional clones based on the semantic matrix. The semantic representation in DeepSim contains syntactic information at the granularity of basic blocks. White et al. [40] fuse information on structure and identifiers from CFG, identifiers and bytecodes, and adopt recurrent neural networks to learn features based on syntactic and semantic analysis. Different from these approaches, we use word embedding to extract AST and graph embedding to extract CFG, then we use the combination of syntactic and semantic features to train a deep learning model.

Tufano et al. [35] uses four different code representation (i.e., identifiers, AST, bytecode, and CFG) for clone clone detection. It uses four code representations to identify the similarity of code pairs separately and calculate the average as the final similarity result. The thought of using AST as syntax representation and CFG as semantics representation is parallel with ours and is very related. Different from Tufano et al., we combine the syntactic and semantic features before using the deep learning model instead of calculating the average. Since considering syntax and semantics, separately may lead to an opposite result, calculating the average directly may not be suitable. Besides, we use Graph2vec instead of other graph embedding techniques for feature representation. Different from HOPE generating vectors for each node, Graph2vec generates a vector for the overall graph. Since calculating the average of node vectors may hide the related among nodes, using Graph2vec to generate a graph vector in more suitable. Furthermore, we extract the call graph to capture the connected methods, and we combine the related methods as a functionality. For each functionality, we analyze the fusion of their syntax and semantics for functional code clone detection.

6 CONCLUSION

We have presented a novel approach to detect functional code clones with code representation generated from the fusion embedding learning of syntactic and semantic information at functionality granularity, and a deep feature learning model that learns the syntactic and semantic features which convert clone detection into a binary classification problem. We have conducted extensive experiments on a large real-world dataset. The results show that our approach achieves a significant advance over state-of-the-art approaches in terms of F1 measure, and it has good detection efficiency after the model training.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for insightful comments. This work is supported partially by National Natural Science Foundation of China(61932012, 61802171), and Fundamental Research Funds for the Central Universities(14380021).

REFERENCES

[1] Brenda S Baker. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*. IEEE, 86–95.

[2] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. 2000. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, 98–107.

[3] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* 33, 9 (2007), 577–591.

[4] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. 2003. Code compaction of matching single-entry multiple-exit regions. In *Proceedings of the 10th International Static Analysis Symposium*. Springer, 401–417.

[5] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 855–864.

[6] Reid Holmes and Gail C Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*. IEEE, 117–125.

[7] Chenping Hou, Feiping Nie, Xuelong Li, Dongyun Yi, and Yi Wu. 2014. Joint embedding learning and sparse regression: A framework for unsupervised feature selection. *IEEE Transactions on Cybernetics* 44, 6 (2014), 793–804.

[8] Sohei Ito. 2018. Semantical equivalence of the control flow graph and the program dependence graph. *arXiv preprint arXiv:1803.02976* (2018).

[9] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE, 96–105.

[10] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. ACM, 55–64.

[11] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. 2014. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188* (2014).

[12] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 664–675.

[13] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[14] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*. Springer, 40–56.

[15] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of 8th Working Conference on Reverse Engineering*. IEEE, 301–309.

[16] Omer Levy, Yoav Goldberg, and Ido Dagan. 2015. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics* 3 (2015), 211–225.

[17] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*. USENIX, 289–302.

[18] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. 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*. ACM, 872–881.

[19] Xing Liu and P Gontey. 1987. Program translation by manipulating abstract syntax trees. In *Proceedings of the C++ Workshop*. 345–360.

[20] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S McKinley. 2015. Does automated refactoring obviate systematic editing?. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 392–402.

[21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[22] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*.

[23] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005* (2017).

[24] Manziba Akanda Nishi and Kostadin Damevski. 2018. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software* 137 (2018), 130–142.

[25] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1105–1114.

[26] J-F Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. 1999. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th International Workshop on Program Comprehension*. IEEE, 49–56.

[27] Daniel Perez and Shigeru Chiba. 2019. Cross-language clone detection by learning over abstract syntax trees. In *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 518–528.

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

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

[30] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 1157–1168.

[31] Abdullah Sheneamer and Jugal Kalita. 2016. Semantic clone detection using machine learning. In *Proceedings of the 15th IEEE International Conference on Machine Learning and Applications*. IEEE, 1024–1028.

[32] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. 1997. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems* 39, 1 (1997), 43–62.

[33] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. 2015. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1055–1090.

[34] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. 2017. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 60–70.

[35] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, 542–553.

[36] Tim A Wagner, Vance Maverick, Susan L Graham, and Michael A Harrison. 1994. Accurate static estimators for program optimization. *ACM Sigplan Notices* 29, 6 (1994), 85–96.

[37] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1225–1234.

[38] Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code.. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 3034–3040.

[39] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of 31st IEEE International Conference on Software Engineering*. IEEE, 364–374.

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

[41] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 783–794.

[42] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 141–151.