DSFM: Enhancing Functional Code Clone Detection with Deep Subtree Interactions

Zhiwei Xu
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Shaohua Qiang
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Dinghong Song
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Min Zhou*
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Hai Wan
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Xibin Zhao
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Ping Luo
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China

Hongyu Zhang
School of Big Data and Software
Engineering
Chongqing University
Chongqing, China

ABSTRACT

Functional code clone detection is important for software maintenance. In recent years, deep learning techniques are introduced to improve the performance of functional code clone detectors. By representing each code snippet as a vector containing its program semantics, syntactically dissimilar functional clones are detected. However, existing deep learning-based approaches attach too much importance to code feature learning, hoping to project all recognizable knowledge of a code snippet into a single vector. We argue that these deep learning-based approaches can be enhanced by considering the characteristics of syntactic code clone detection, where we need to compare the contents of the source code (e.g., intersection of tokens, similar flow graphs, and similar subtrees) to obtain code clones. In this paper, we propose a novel deep learning-based approach named DSFM, which incorporates comparisons between code snippets for detecting functional code clones. Specifically, we improve the typical deep clone detectors with deep subtree interactions that compare every two subtrees extracted abstract syntax trees (ASTs) of two code snippets, thereby introducing more fine-grained semantic similarity. By conducting extensive experiments on three widely-used datasets, GCJ, OJClone, and BigCloneBench, we demonstrate the great potential of deep subtree interactions in code clone detection task. The proposed DSFM outperforms the state-of-the-art approaches, including two traditional approaches, two unsupervised and four supervised deep learning-based baselines.

CCS CONCEPTS

• **Software and its engineering → Reusability**; **Maintaining software**; • **Computing methodologies → Machine learning**.

KEYWORDS

Code clone detection, semantic clone, code similarity, factorization machine

ACM Reference Format:
Zhiwei Xu, Shaohua Qiang, Dinghong Song, Min Zhou, Hai Wan, Xibin Zhao, Ping Luo, and Hongyu Zhang. 2024. DSFM: Enhancing Functional Code Clone Detection with Deep Subtree Interactions. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639215

1 INTRODUCTION

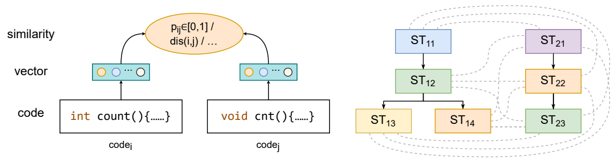
Detecting code clones is critical in software engineering. It was shown to be very useful in code search and reuse [10, 16], software refactoring [29] and bug detection [13, 29]. Functional code clones are produced by copy-paste operations, or when a developer implements a certain functionality similar to an existing one [36]. Many such code duplicates may introduce inconspicuous defects to software, making software hard to develop and maintain [25].

Functional code clones refer to code snippets that have similar functionality. Their syntax can be similar or dissimilar, posing a severe challenge to the existing code clone detectors. Basically, previous efforts follow two typical perspectives - traditional approaches [23, 24, 39, 44, 58] and recent deep learning-based approaches [14, 15, 20, 31, 33, 38, 46–48, 50, 53, 57]. Traditional approaches are recognized to be ineffective in detecting functional code clones [28]. They generally utilize common comparison metrics, e.g., edit distance [23, 44], number of shared tokens [39] and Weisfeiler-Lehman Graph Kernel [58], to measure the distance between two code snippets. The disadvantage of them is obvious: they are only capable of handling Type-1 and Type-2 clones (i.e., code

[*] Min Zhou is the corresponding author (mzhou@tsinghua.edu.cn).

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639215



(a) Previous deep learning-based methods. (b) Our subtree-based method.

Figure 1: Comparison of previous deep learning-based methods and our subtree-based methods. Instead of comparing the representations of two code snippets directly, our subtree-based method first consider two ASTs as two sets of subtrees, and explicitly compare these subtrees to yield the similarity.

snippets with similar syntax) [23, 24, 58], or at most Type-3 clones (i.e., code snippets who have a few different statements) [39, 44].

To further improve the performance of code clone detection, particularly for Type-4 clones (i.e., code snippets who have dissimilar syntax), recent studies [14, 15, 20, 31, 33, 38, 46–48, 50, 53, 57] exploit deep learning techniques to represent each code snippet as a vector (a.k.a., distributed representation). The code vector can be generated by considering the code snippet as semantic tokens [48], flow graph [57], or abstract syntax tree (AST) [20, 46, 47, 50, 53], and applying effective neural models such as recursive neural network [47, 53], recurrent neural network [47, 48, 53], TreeLSTM [20, 46], tree-based convolution [50] and graph neural network [14, 15, 20, 31, 33]. Recent evidence [27, 28] reveals that the quality of code vectors is of great importance in code clone detection, and the leading approaches [14, 48, 53] all spare no effort to improve code representation models.

Considering the importance of code representation learning, a question naturally arises: *Is representing code snippets as single vectors all we need for unleashing the potential of clone detectors?* One motivation for this question is that code clone detection has always been done by comparing specific contents (e.g., intersection of tokens [24, 39, 44], similar flow graph [58], and similar subtrees [5, 23]) of code snippets [9]). This comparison paradigm is technically explainable, which aligns well with the intuitive solutions for the code similarity measurement task. Unexpectedly, although representing code snippets as single vectors makes it possible for deep learning-based approaches to handle Type-4 clones, the widely-accepted traditional comparison paradigm has been left behind. Figure 1(a) shows a typical example of deep learning-based clone detectors. They first project the information of code snippets (e.g., code structures and code semantics) into single vectors and then use the distances between these vectors for code similarity measurement. Despite introducing various kinds of code information, the explicit comparisons between the contents of code snippets has been overlooked. This poses a challenge to the quality of code vectors - they are required to compensate for the benefit of the widely-accepted comparison paradigm.

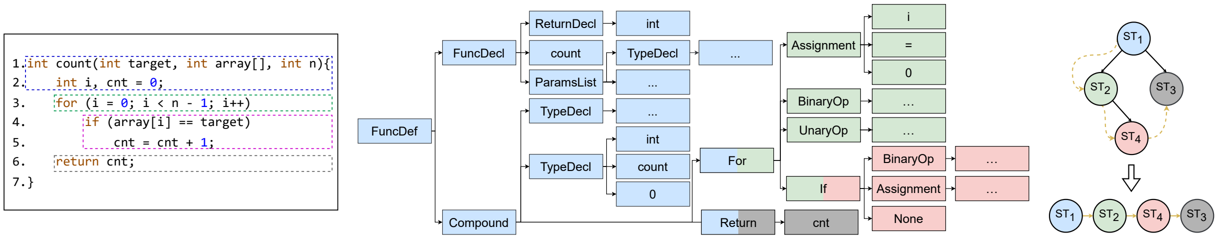
Another motivation for the question is that there exists potential information loss when deep learning-based techniques represent code snippets as single vectors. It is noted that most of the well-acknowledged code representation models [2, 4, 32, 53] generate a code vector through synthesizing a collection of preprocessed objects (e.g., paths [4], subtrees [53], and AST nodes [2, 32]). A pooling layer is commonly utilized to fuse the embeddings of candidate objects into a single code vector, while only a portion of the information (e.g., maximum or mean values) is allowed to be retained [2, 32, 53]. We call this the *refinement* of the preprocessed code information. Despite preserving the desired signals, such refinement could easily lead to the loss of favorable preprocessed information. To illustrate, we can analogize the refinement of code information to the image downsizing in the field of image processing, where the resolution of images decreases and the images become blurred [1, 22]. Therefore, the pooling causes potential information loss when representing code snippets as single vectors. An effective approach would be also to exploit the information preceding the pooling to measure the similarity of code snippets, rather than only using the refined code information.

In this paper, we address the limitations faced by existing deep learning-based approaches by introducing *deep subtree interactions*. Instead of hoping single code vectors directly complete all aspects of functional code clone detection, we explicitly consider each code snippet as a collection of blocks and apply comparisons between the representations of these blocks. Our approach is driven by the success of traditional subtree-based detectors such as Deckard [23] and the work in [5]; we also incorporate powerful deep learning techniques for detecting Type-4 code clones. That is, we represent each block of code snippets as vectors by using the syntactic and semantic features of its corresponding subtree. The interactions (or “comparisons” interchangeably) between deep subtrees is enabled to realize an effective functional code clone detector.

Specifically, we present Deep Subtree Factorization Machine (DSFM), a novel approach that introduces deep subtree interactions for detecting functional code clones. Inspired by factorization machines [17, 35], we model the functional similarity of code snippets at both the program level and the block level. Program-level similarity measures the holistic similarity between two code snippets, similar to the typical deep learning-based clone detectors [14, 15, 20, 31, 33, 38, 46–48, 50, 53, 57]; whereas block-level similarity provides a more granular similarity of every two blocks of code snippets to enhance functional code clone detection. The combination of these two techniques ensures the generalizability of our DSFM. The workflow of our DSFM is as follows. It first transforms each code snippet into an AST and performs preorder traversal to extract subtree sequence[1] from the AST (see Figure 2). Further, a recursive and a recurrent encoder is applied to introduce syntactic information within and outside each subtree into their representations. These representations are fed into a scoring layer to measure the program-level and the block-level similarity, which ultimately derives the overall similarity for functional code clone detection.

We carry out experiments on three widely-used datasets: GCJ [31, 48, 57], OJClone[14, 32, 46, 50, 53, 56], and BigCloneBench[31, 38, 39, 42, 46, 48, 50, 53, 56, 57]. The results show that our model is more effective against the state-of-the-art. Our DSFM achieved F1 scores of 97.3%, 99.6%, and 96.5% on GCJ, OJClone, and BigCloneBench, respectively, which are 2.87%, 1.04%, and 0.53% higher than the highest F1 scores achieved by the baselines. This strong evidence

[1] Subtree sequence is an ordered collection of subtrees that is generally used [40, 53] to capture the statistical naturalness of code [18, 34].



```
int count(int target, int array[], int n){
    int i, cnt = 0;
    for (i = 0; i < n - 1; i++)
        if (array[i] == target)
            cnt = cnt + 1;
    return cnt;
}
```

(a) Code snippet.

(b) Abstract syntax tree.

(c) Subtree sequence.

Figure 2: Example of a code snippet, along with its abstract syntax tree and subtree sequence. The abstract syntax tree is partitioned into several subtrees using Algorithm 1. Subtrees are marked with blue, green, pink and gray color, respectively.

demonstrates that the proposed deep learning-based comparison paradigm is generic, which might enable new possibilities for the community of functional code clone detection.

The contributions of this paper are listed as follows:

- We motivate and present a new code clone detection paradigm: incorporating deep learning techniques and explicit comparisons between contents of code snippets to detect functional code clones.
- Based on the above paradigm, we propose and implement a novel approach named DSFM, which introduces deep subtree interactions to improve the effectiveness of functional code clone detection.
- The proposed approach DSFM is evaluated on three widely-used datasets, and the results demonstrate that DSFM outperforms the baselines and the state-of-the-art.

The rest of the paper is organized as follows. Section 2 introduces the background. Section 3 presents the proposed approach DSFM. Section 4 reports the experiments. Section 5 discusses the threats to this work. Section 6 summarizes the related work. Section 7 concludes this paper.

2 BACKGROUND

2.1 Types of Code Clones

It is widely accepted [6, 37, 42, 48] that code clones can be divided into the following four types:

- **Type-1 (*textual similarity*)**: identical code snippets apart from difference in comments, white space and layout.
- **Type-2 (*lexical similarity*)**: identical code snippets except for different identifier names and literal values, along with Type-1 clone differences.
- **Type-3 (*syntactic similarity*)**: syntactically similar code snippets that differ at the statement level. The code snippets contain statements added, removed and/or modified with regard to each other, along with Type-1 and Type-2 clone differences.
- **Type-4 (*semantic similarity*)**: syntactically dissimilar code snippets that implement the same functionality.

The boundary between Type-3 and Type-4 clones is often ambiguous [45] and cannot be quantitatively defined [15]. In this paper, we refer to near-miss clones in SourcererCC [39] as Type-3 clones. We use Type-2, Strongly Type-3, Moderately Type-3, and Weakly Type-3/Type-4 clones in BigCloneBench [42] to denote no-difference Type-3, low-difference Type-3, medium-difference Type-3, and semantic clones[45], respectively.

2.2 Abstract Syntax Tree and Its Deep Modeling

Programming language is highly structured. To capture the syntactic information of source codes for deep learning models, a popular method is to transform code snippets into ASTs and encode the structure of ASTs into code representations [4, 15, 32, 50, 53]. Figure 2(b) illustrates an example of an AST. We can observe that each leaf nodes represent a token of the code snippet and they are organized in a tree structure to reflect code syntax.

To incorporate ASTs into deep learning techniques, researchers [21, 41, 53] present recursive neural network (RvNN) over trees to generate tree embeddings. Let $\mathbf{z}_n$ be the embedding of a node n, the computation of RvNN for n in this work is as follows:

$$\mathbf{h}_n = \alpha(\mathbf{W}\mathbf{z}_n + \sum_{c \in C(n)} \mathbf{h}_c + \mathbf{b}) \tag{1}$$

where $\mathbf{W}$ and $\mathbf{b}$ represent a fully-connected layer, $C(n)$ represents the child nodes of n, and $\alpha()$ represents the activation function such as tanh and ReLU. The computation is formalized recursively and performs a bottom-up processing.

2.3 Factorization Machines

Factorization machine (FM) [35] is a well-known solution for learning feature interactions in the field of information retrieval. Given a one-hot feature vector $x \in \{0, 1\}^n$, FM estimates the target as:

$$\hat{y}(x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} w_{ij} x_i x_j \tag{2}$$

where w_0 is the global bias, w_i represents the strength of the i-th variable, and w_{ij} models the interaction between the i-th variable and the j-th variable.

Unfortunately, the effectiveness of FM is limited due to its belonging of multivariate linear model family [55]. To improve the generalizability of FM, neural factorization machine (NFM) [17]

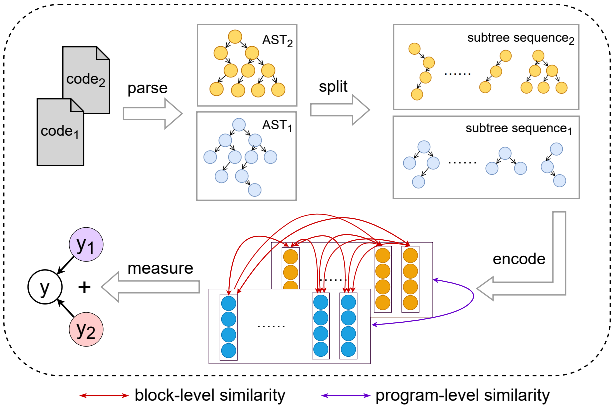


Figure 3: Overview of our DSFM. Code snippets are first parsed into ASTs, which are decomposed into subtree sequences. We adopt the subtree encoder to represent them as vector representations, and apply program-level and block-level similarities to derive the overall functional similarity.

is proposed to combine the strengths of FMs and deep neural networks. It can be calculated as:

$$\hat{y}(x) = w_0 + \sum_{i=1}^{n} w_i x_i + f(\sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbf{v}_i x_i \odot \mathbf{v}_j x_j) \qquad (3)$$

where $\odot$ is the inner product, $\mathbf{v}_i \in \mathbb{R}^d$ is the embedding of the i-th variable, d is the size of embedding vector, and $f()$ is the stacked fully-connected layer. Note that the second-order term can be reformulated as $\frac{1}{2}[(\sum_{i=1}^{n} \mathbf{v}_i x_i)^2 - \sum_{i=1}^{n}(\mathbf{v}_i x_i)^2]$, whose time complexity is $O(nd)$ (see [35] for proof details).

3 PROPOSED APPROACH

Figure 3 explains the overall architecture of the proposed approach DSFM. First, it parses code snippets into ASTs via existing parsers and adopts preorder traversal to convert these ASTs into subtree sequences. Then, it applies a bottom-up recursive encoder and a recurrent sequence encoder to represent each subtree as a vector. By introducing program-level and block-level similarity, it derives the overall similarity of two code snippets. At last, a binary classification problem is formulated for detecting code clones.

3.1 Generating Subtree Sequence via Preorder Traversal

To capture the syntactic features of code snippets, it is common to transform code snippets into ASTs. Further, our DSFM performs a preorder traversal on ASTs to convert ASTs into subtree sequences, which are utilized for measuring the block-level functional similarity. It is important to note that, in order to align with the code clone detection task, the subtree sequence extracted by our approach is completely different from previous approaches [40, 53]. DSFM pays attention to subtrees that corresponds to blocks of code snippets while the previous approaches extract statement subtrees for representation purpose. In the last observation of Section 4.4.2, we will illustrate that our subtree design is more effective in code similarity measurement task.

Algorithm 1: Extract Subtree Sequence from an AST

```
    Input: Root of an AST $head$, Set of delimiters $D$;
    Output: Subtree sequence $S$;
1   Function extract($root$)
2       Construct a subtree node $s$ based on $root$;
3       if $root$ is the topmost node || $root$ not in $D$ then
4           foreach $child$ in $root$ do
5               Add extract($child$) as the child of $s$;
6           end
7       end
8   return $s$
9   Function traverse($root$)
10      if $root$ in $D$ then
11          append extract($root$) to $S$;
12      end
13      foreach $child$ in $root$ do
14          traverse($child$);
15      end
16  Initialize $S \leftarrow []$;
17  traverse($head$);
18  return $S$;
```

Algorithm 1 illustrates the algorithm for extracting subtree sequence from an AST. It takes the root node of an AST *head* and a set of delimiters D as input and outputs the extracted subtree sequence S. The set of delimiters D can decide the granularity of subtrees (or blocks), which is predefined according to the target programming language. In our work, D normally comprises *If*, *For*, *While*, *Switch*, *ClassDeclaration*, and *FuncCall*, etc. The basic idea behind the selection of delimiters is that these delimiters can have better expressions about the intended semantics intuitively.

The algorithm consists of two functions, i.e., `extract` for extracting a subtree based on the current node and `traverse` for preorder traversing the AST to find the delimiter nodes. In function `extract`, it accepts a node named *root* as input (line 1) and constructs a tree node object s of it (line 2). Then, if node *root* is not the topmost node (not from line 11) or is not a delimiter node (line 3), we recursively perform `extract` for each child and add the result as current tree node s's child (lines 4-6). At last, a subtree extracted from node *root* is returned (line 8). In the function `traverse`, it takes a node named *root* as input (line 9). If this node is a delimiter node (line 10), we perform `extract` to generate a subtree based on this node and append it to the subtree sequence S (line 11). `traverse` continues to traverse the AST and find the next delimiter node to be extracted (lines 13-15). In the main algorithm body, we initialize an empty subtree sequence S (line 16) and use `traverse` to extract subtrees (line 17) as the final result (line 18). We note that a subtree includes all the delimiter nodes used to identify its boundary (refer to the different colors for subtrees in Figure 2(b)), which intactly provides information of each subtree for subsequent processing.

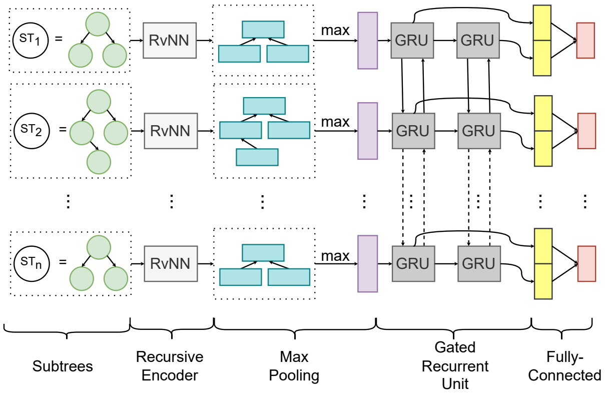


Figure 4: Architecture of subtree encoder. The subtrees are first fed into recursive neural network to capture structural knowledge and then fed into gated recurrent unit to capture sequential knowledge.

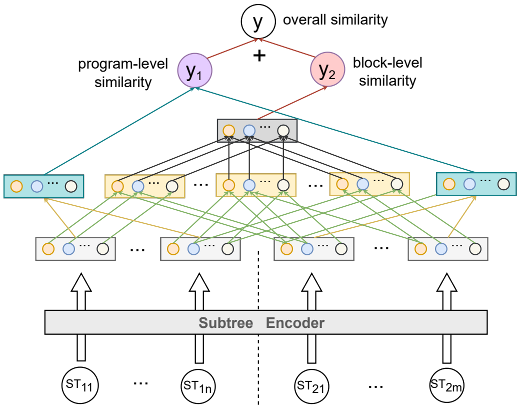


Figure 5: Proposed network for measuring code functional similarity, which combines both program-level and block-level similarity to detect functional code clones.

3.2 Encoding Subtrees with Recursive and Recurrent Network

We design a two-phase *subtree encoder* module to learn the representation of subtrees, as shown in Figure 4. In the first phase, a recursive neural network (RvNN) followed by a max pooling layer is applied to encode each subtree. In the second phase, we use a two-layer Gated Recurrent Unit (GRU) to capture the sequential relationship among those subtrees.

3.2.1 Recursive Encoder for Structural Knowledge. Since each subtree extracted from the AST is a form of the tree structure, we use Recursive Neural Network (RvNN) [41] to model the structural knowledge. Prior to that, DSFM defines an embedding vocabulary $\mathbf{W}_n \in \mathbb{R}^{|T| \times d}$. Let x_n be the one-hot vector of node n, its embedding $\mathbf{z}_n$ is looked up from the corresponding row as follows:

$$\mathbf{z}_n = \mathbf{W}_n \mathbf{x}_n \tag{4}$$

Then, for the root node of every subtree of the subtree sequence, DSFM employs RvNN for capturing structural knowledge. The embeddings of node n is thus updated as:

$$\mathbf{h}_n = \text{RvNN}(\mathbf{z}_n) \tag{5}$$

where RvNN details in Equation (1) and the activation function used in this work is identity mapping. Denote s_i as i-th subtree of the subtree sequence, we apply max pooling to produce the subtree vector $\mathbf{q}_i$ as:

$$\mathbf{q}_i = \max\{\mathbf{h}_n | n \in \mathcal{N}(s_i)\} \tag{6}$$

where $\mathcal{N}(s_i)$ is a collection of all nodes in subtree s_i. Max pooling preserves the maximum values of node vectors and generates the embeddings of each subtree. We then could represent the subtree sequence as $\{\mathbf{q}_1, \mathbf{q}_2, ..., \mathbf{q}_n\}$. Note that the pooling here is necessary, which serves as an aggregator for information from excessive amount of nodes and generates the embeddings of each subtree for comparisons. Comparing the embeddings of these nodes is of little help for detecting functional code clones, as directly using the information of nodes is proven to be ineffective [53]. The blocks (or subtrees) can better reflect the code semantics.

3.2.2 Recurrent Encoder for Sequential Knowledge. After modeling the structural knowledge for subtrees, we adopt a recurrent encoder to capture the sequential relationship among those subtrees. The family of recurrent encoders consists of many variants such as Long Short-Term Memory (LSTM) [19] and Gated Recurrent Unit (GRU) [11]. In this work, we adopt GRU rather than LSTM because GRU can achieve similar performance with fewer parameters [53]. Based on the representation of subtree sequence $\{\mathbf{q}_1, \mathbf{q}_2, ..., \mathbf{q}_n\}$, a bidirectional GRU is used to introduce the sequential knowledge from both normal and opposite order. The formula is as follows:

$$\mathbf{g}_i = [\overrightarrow{\text{GRU}}(\mathbf{q}_i); \overleftarrow{\text{GRU}}(\mathbf{q}_i)] \tag{7}$$

where $[;]$ represents the concatenation of two vectors; $\overrightarrow{\text{GRU}}$ and $\overleftarrow{\text{GRU}}$ performs GRU upon the original and reverse subtree sequence, respectively. Then, we adopt a fully-connected layer to fuse the two-way outputs:

$$\mathbf{v}_i = \alpha(\mathbf{W}_f \mathbf{g}_i + \mathbf{b}_f) \tag{8}$$

where $\mathbf{W}_f \in \mathbb{R}^{d \times 2d}$, $\mathbf{b}_f \in \mathbb{R}^d$ are weight matrix and bias term. The activation function $\alpha()$ is identity mapping in this work.

3.3 Measuring Functional Similarity of Code

After introducing structural and sequential knowledge into subtrees, we present a scoring layer to measure both program-level and block-level similarity to detect functional code clones. Program-level similarity measures the similarity between the representations of two code snippets, similar to typical deep learning-based clone detectors [14, 15, 20, 31, 33, 38, 46–48, 50, 53, 57]. Block-level similarity measures the similarity between every two blocks of two code snippets and is reflected by the similarity between subtrees of two ASTs. They work together for the effectiveness of DSFM.

Specifically, as depicted in Figure 5, given two code snippets, we first use the subtree encoder to generate the subtree representations of them, i.e., $\mathcal{V}_1 = \{\mathbf{v}_{11}, \mathbf{v}_{12}, ..., \mathbf{v}_{1n}\}$ and $\mathcal{V}_2 = \{\mathbf{v}_{21}, \mathbf{v}_{22}, ..., \mathbf{v}_{2m}\}$, where $\mathbf{v}_{ij}$ represents the embedding of j-th subtree from i-th code snippet and $\mathcal{V}_i$ represents the set of subtree embeddings of i-th code snippet.

The program-level similarity (also the first-order term) is designed to compare the code snippets from a holistic perspective. It represents the entire code snippets as single representations, which are leveraged for program-level similarity. To acquire the single representation for each code snippet, we use max pooling to incorporate the information of all subtrees. Note that the pooling here does not introduce additional information loss due to the existence of the block-level similarity. In contrast, the program-level similarity derived from this pooling can be an effective complement to the block-level similarity. Taking the first code snippet $\mathcal{V}_1$ as an example, the code vector $\mathbf{c}_1$ ($\mathbf{c}_2$ is similar) is calculated as:

$$\mathbf{c}_1 = \max\{\mathbf{v}_{1i} | i = 1, 2, ..., n, \mathbf{v}_{1i} \in \mathcal{V}_1\} \tag{9}$$

where max{} is the element-wise max pooling. Then, the first-order term can be derived using the inner product as:

$$\hat{y}_1 = \mathbf{c}_1^T \mathbf{c}_2 \tag{10}$$

The block-level similarity (also the second-order term) is designed to measure the functional similarity of every two blocks of code snippets, providing a fine-grained perspective for detecting code clones. Considering blocks are converted into subtree vectors using the previous designs, we introduce block-level similarity based on $\mathcal{V}_1$ and $\mathcal{V}_2$. Different from the existing deep learning-based clone detectors, our DSFM conducts explicit interactions between the embeddings of every two subtrees and derive the block-level similarity from these interactions. The functional similarities of subtrees are accumulated by summation and a fully-connected layer are further applied, which is calculated as follows:

$$\hat{y}_2 = \mathbf{W}_l(\sum_{i=1}^{n}\sum_{j=1}^{m} \mathbf{v}_{1i} \odot \mathbf{v}_{2j}) + \mathbf{b}_l \tag{11}$$

where $\mathbf{W}_l \in \mathbb{R}^{1\times d}$ is the weight matrix, $\mathbf{b}_l \in \mathbb{R}^1$ is the bias, and $\odot$ is the element-wise product between two vectors. In this way, each subtree vector of $\mathcal{V}_1$ has a functional similarity with each subtree vector of $\mathcal{V}_2$, which provides a measure of functional similarity between two code snippets from the block-level perspective. By integrating both the first-order term and the second-order term, DSFM derives the overall functional similarity as follows:

$$\hat{y} = \hat{y}_1 + \hat{y}_2 \tag{12}$$

where $\hat{y}$ is the overall functional similarity between two code snippets measured by our DSFM.

3.4 Optimization

Following the above steps, the predicted similarity can be calculated. We use Cross Entropy Loss to optimize our DSFM:

$$J(y, \hat{y}) = -\sum_i (y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)) \tag{13}$$

where $y \in \{0, 1\}$ refers to the ground truth, and $\hat{y} \in [0, 1]$ refers to the predicted probability, the higher the more likely it is to be a clone pair.

Table 1: Statistics of datasets.

Dataset	GCJ	OJClone	BigCloneBench
Language	Java	C	Java
Granularity	File	File	Method
Label	Automatic	Automatic	Manual
# of functionalities	12	15	10
# of code snippets	1,664	9,975	9,143
Avg./Max.			
# of tokens	360/2,025	250/2,353	227/16,253
# of subtrees	10/74	15/193	6/288
AST depth	16/34	14/39	10/235
AST width	56/656	65/498	40/4,164

3.5 Analysis of Computation Complexity

The overall computation complexity of DSFM is $O(BNWD)$, where B is the batch size, N is the number of subtrees of AST, W is the average width of subtree, and D is the average depth of subtree. Here, the time complexity of vector operations is ideally ignored, as the presence of acceleration techniques of deep learning frameworks such as TensorFlow and PyTorch makes it difficult to measure. The complexity analysis is conducted from a macro perspective.

Specifically, the overall computation complexity of our approach mainly consists of three parts, $O(BNWD)$ for the recursive encoder, $O(BN)$ for the recurrent encoder, and $O(BN)$ for the scoring layer. For the recursive encoder, DSFM first collects all nodes of the same depth in a batch of data samples, with a time complexity of $O(BNW)$, and performs a bottom-up recursion to encode the collected nodes, with a time complexity of $O(D)$. The recurrent encoder costs a complexity of $O(B)$ to collect subtrees at the same length in a batch of data samples and takes a time complexity of $O(N)$ to perform recurrent operations subsequently. The computation complexity of the scoring layer mainly comes from the block-level similarity. However, from the analysis in Section 2.3, we could derive the time complexity is $O(BN)$, linear to the number of subtrees. In summary, we can conclude that the overall computation complexity of our approach is $O(BNWD) + O(BN) + O(BN) = O(BNWD)$, where NWD is approximately equal to the AST size.

4 EXPERIMENT

In this section, we aim to answer the following research questions:

RQ1: How effective is our DSFM on different datasets (also on different types of clones) compared with the baseline approaches?

RQ2: What is the efficiency of our DSFM?

RQ3: How does our DSFM perform with alternative settings?

4.1 Experimental Setup

4.1.1 Datasets. We choose three credible datasets for evaluating our DSFM and the baselines: GCJ, OJClone, and BigCloneBench. Table 1 provides the statistics of three datasets.

GCJ is built based on Google Code Jam competition dataset, with 1,664 projects from 12 different competition problems (functionalities). Following the procedure of previous literature [57], we refer to each entire solution as a code snippet and build GCJ by considering the code snippets within the same functionalities as clones; otherwise, non-clones. It is manually verified that most of clone pairs of GCJ belong to Type-4 clones (syntactically dissimilar) [57]. We randomly select 1,000, 332, and 332 code snippets for the training, validation, and test set, respectively. Lastly, GCJ creates nearly 0.6 million code pairs where clone pairs account for 19.6%.

OJClone is constructed using the same method proposed in TBCCD [50]. The original OJClone consists of 104 programming problems together with the source code snippets that students submitted as the solutions [32]. In OJClone, almost all code snippets have dissimilar syntax, resulting in its clone pairs mostly being Type-4 clones. Similar to [46, 50, 53], we select 500 solutions from each of the first 15 questions to build OJClone. We randomly choose 500, 500, 6,500 solutions for the validation set, testing set, and training set, respectively. Two different code snippets for solving the same programming problem form a clone pair; otherwise, they form a non-clone pair. Therefore, it will produce 124,750 code pairs in the validation and testing set and 21,121,750 pairs in the training set which is time-consuming for evaluation. Same as [50], we sample around one million code pairs as the training set, where clone pairs account for 6.6%.

BigCloneBench [42] is a popular benchmark dataset that was built by mining at first and then manually verified by experts. BigCloneBench contains over 6 million tagged clone pairs and 260 thousand false pairs collected from 25 thousand systems in an inter-project Java repository. The code clones from BigCloneBench have been tagged with different clone types, including 0.5% of Type-1 clones, 0.1% of Type-2 clones, 0.2% of Strongly Type-3 clones, 1.0% of Moderately Type-3 clones, and 98.2% of Weakly Type-3/Type-4 clones. Just as the authors of CDLH [46] and TBCCD [50] do for constructing BigCloneBench, we randomly select 500, 500, and 8,134 code snippets for the validation set, test set, and the training set. This pipeline provides us 124,750 code pairs in the validation and test set and 33,068,778 code pairs in the training set. Similarly, we randomly sample approximately 1 million code pairs where true clone pairs account for 14.0%.

4.1.2 Baselines. Two traditional approaches (i.e., Deckard [23] and SourcererCC [39]), two unsupervised deep learning-based approaches (i.e., DLC [47] and InferCode [8]) and four supervised functional code clone detectors (i.e., code2vec [4], code2seq [3], TBCCD [50], and ASTNN [53]) are considered as the baselines. **Deckard** is a traditional subtree-based approach, which partially motivates the design of our approach. **SourcererCC** is a leading traditional approach that exploits tokens to explore Type-3 clones. We choose these two baselines to illustrate the difficulty of traditional approaches in Type-4 clones. **DLC** is a pioneering approach that applies deep learning techniques in very early years. **InferCode** is a code pre-training approach and is chosen as a complement to unsupervised approaches. The introduction of these two representative unsupervised approaches aims to show the performance achieved by using only the information of deep features. **code2vec** and **code2seq** are code representation approaches, which are introduced as baselines to reveal the performance of typical deep learning techniques in detecting functional code clones. We also compare our DSFM with TBCCD and ASTNN. **TBCCD** is a supervised approach designed specially for functional code clone detection. It also drives the construction of OJClone and BigCloneBench datasets in this work. **ASTNN** is a subtree-based neural network similar to our DSFM, which is introduced to validate whether the crafted designs of our approach (e.g., subtree construction, block-level subtrees and deep subtree interactions) are effective.

We have also considered other approaches as our baselines, such as CDLH [46], DeepSim [57], word2vec+graph2vec [14], SCDetector [48], FCCA [20], HOLMES [31] and code2vec+graph2vec [52]. But these approaches have not been made public by their authors, or have limitations that prevent them from participating in comparisons on customized datasets (see detailed illustration in Section 5). According to the results in their original papers, under the same/similar experimental settings, none of the above approaches has an F1 score higher than 76% on GCJ, 96% on OJClone, and 82% on BigCloneBench.

4.1.3 Experimental Settings. In our experiments, the model that achieves the best performance on the validation set is used to evaluate on the testing set. The parameters of all baselines are set to default values, common values, or tuned to be optimal for fairness. Specifically, for Deckard, two code snippets with enough percentage of similar lines are considered as a clone pair, where the threshold is tuned using the validation set. For DLC, the number of unsupervised training iterations is set to 10. The head of code2vec and code2seq is defined as the inner product as it is proven to be simple but effective [30]. For InferCode, we follow the same unsupervised way introduced by their paper to detect code clones, and the cosine similarity threshold is set to 0.8, the same as their released source code. For our DSFM, the training epochs is set to 5 and the learning rate is set to 0.001, optimized using Adam optimizer [26]. The embedding dimension of our DSFM is 128. There are many parsers designed for different types of programming languages to transform code snippets into ASTs, such as javalang[2], pycparser[3], srcml [12] and fAST [51]. In this work, we use javalang for Java and pycparser for C. All our experiments are carried out on a platform with a 12-core CPU, 128 GB memory, and a RTX 3090 GPU.

4.1.4 Evaluation Metrics. We choose precision, recall and F1 score as metrics similar to [14, 15, 20, 24, 31, 33, 38, 39, 44, 46–48, 50, 53, 57]. Precision is the fraction of predicted code clones that are code clones. Recall is the fraction of code clones that are correctly identified. F1 score is the harmonic mean of precision and recall.

4.2 RQ1: Performance Comparison

We compare the proposed approach with 8 baselines. Table 2 shows the overall performance of DSFM and the baselines on three datasets. The best results are highlighted in bold.

From Table 2, we observe that the proposed DSFM has the most superior performance in F1 score on all three datasets, i.e., it achieves an F1 score of 97.3% on GCJ, 99.6% on OJClone, and 96.5% on BigCloneBench. The F1 scores achieved by DSFM on each dataset are higher than the highest F1 scores achieved by the baselines, achieving a boost of 2.87% on GCJ, 1.04% on OJClone, and 0.53% on BigCloneBench. Particularly on GCJ, we note that our DSFM achieves an obvious improvement in F1 score compared

[2] https://github.com/c2nes/javalang
[3] https://github.com/eliben/pycparser

Table 2: Overall performance of competing approaches on three datasets.

Method	**GCJ**			**OJClone**			**BigCloneBench**		
	Precision	**Recall**	**F1 score**	**Precision**	**Recall**	**F1 score**	**Precision**	**Recall**	**F1 score**
Deckard†	70.5	5.5	10.3	64.0	2.2	4.3	16.1	28.8	20.7
SourcererCC†	75.4	6.7	12.3	26.0	46.8	33.4	96.6	1.8	3.5
DLC‡	20.9	80.6	33.2	6.7	86.2	12.4	15.5	98.6	26.8
InferCode‡	30.1	55.3	38.9	17.9	49.2	26.2	16.6	74.1	27.1
code2vec	60.7	94.3	73.8	32.0	93.4	47.4	39.5	**98.3**	56.4
code2seq	76.5	**95.6**	85.0	76.5	**99.3**	86.4	42.4	92.8	58.2
TBCCD	95.5	93.5	94.5	97.3	98.8	98.0	**97.1**	94.9	96.0
ASTNN	94.8	92.8	93.8	98.6	98.5	98.6	98.1	93.5	95.8
DSFM	**99.4**	95.3	**97.3**	**99.7**	98.5	**99.6**	96.3	96.7	**96.5**

† Deckard and SourcererCC are traditional approaches and cannot handle Type-4 clones.
‡ DLC and InferCode are unsupervised deep learning-based approaches and do not have an end-to-end detection pipeline.

Table 3: Results of competing approaches in F1 score on various types of clones using BigCloneBench.

Method	**T1**	**T2**	**ST3**	**MT3**	**WT3/T4**
Deckard	100	100	78.3	69.6	20.0
SourcererCC	100	100	92.3	52.1	1.4
DLC	99.5	100	96.3	98.0	26.4
InferCode	100	100	100	97.4	26.6
code2vec	100	100	100	100	55.9
code2seq	100	100	100	100	57.8
TBCCD	100	100	100	99.7	95.9
ASTNN	100	100	100	98.9	95.7
DSFM	100	100	100	100	96.4

with the baselines, showing its effectiveness for detecting functional code clones partially. Although ASTNN achieves a very high F1 score on OJClone, its weakness on GCJ is significant. TBCCD achieves outstanding performance on two Java datasets (i.e., GCJ and BigCloneBench), whereas its performance fails to dominate on C dataset (i.e., OJClone). Therefore, none of the baselines can achieve the same stable performance as our approach. Additionally, precision may be more important in practice, revealing the acceptance of the tools, i.e., how many of the code clones found are indeed code clones. It can be seen that DSFM can achieve the highest precision on both GCJ and OJClone and achieve a competitive precision on BigCloneBench. Generally, a precision of 95% is sufficient for users. On GCJ, TBCCD and ASTNN obtain a precision near 95%, indicating that 1-in-20 data instances are false positives that may distract users. Our DSFM reduces it to 1-in-200 false positives, thus greatly improves the productivity of detecting functional code clones for users. All the above results demonstrate that our DSFM is more superior to the baselines.

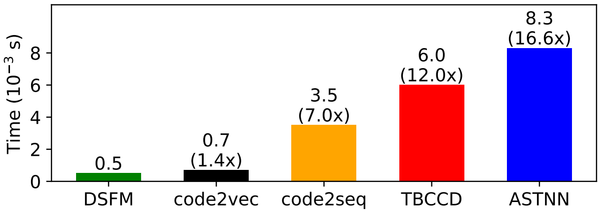


Figure 6: Average computation time required by supervised deep learning approaches for predicting one data instance.

To dig deeper into detection performance, we evaluate the competing approaches on various types of clones using BigCloneBench, which has already been labeled with clone types [42]: Type-1 (T1), Type-2 (T2), Strongly Type-3 (ST3), Moderately Type-3 (MT3) and Weakly Type-3/Type-4 (WT3/T4) (see Section 2.1 for the definitions). We follow the same evaluation procedure as [53, 57], and the F1 score of detectors on each type are reported in Table 3. We observe that DSFM can achieve the highest F1 score on all types of clones compared with the baselines. DSFM realizes an F1 score of 100% on T1, T2, ST3, and MT3, and an F1 score of 96.4% on WT3/T4. Especially on MT3 and WT3/T4, the performance of DSFM over the baselines is apparent, showing that our approach is capable of handling syntactically dissimilar code clones. The advantages of our DSFM mainly come from MT3 and WT3/T4. On MT3, only code2vec and code2seq manage to attain a perfect F1 score of 100%; on WT3/T4, none of the baselines can surpass an F1 score higher than 57.8%, except TBCCD and ASTNN. An interesting phenomenon is that two approaches (i.e., TBCCD and ASTNN) that achieve a high F1 score on WT3/T4, however, fail to reach an F1 score of 100% on MT3. This phenomenon may be due to their tendency to narrow the discriminative boundary for the purpose of generalizability.

To conclude, the above experiments demonstrate that the proposed approach is more effective compared with the baselines, which unveils the significant potential of deep learning-based comparison paradigm for detecting functional code clones.

4.3 RQ2: Efficiency Evaluation

To illustrate the efficiency, we investigate the computation time required for a model to predict one data instance, whose reciprocal

Table 4: Performance of our DSFM trained and tested on different sets.

Dataset		DSFM		
Training set	**Test set**	**Precision**	**Recall**	**F1 score**
OJClone	GCJ	98.8	92.5	95.5
BigCloneBench	GCJ	95.7	89.8	92.7
GCJ	OJClone	98.7	98.8	98.8
BigCloneBench	OJClone	98.4	97.6	98.0
GCJ	BigCloneBench	96.0	95.1	95.5
OJClone	BigCloneBench	98.8	93.9	96.3

is commonly referred to as the prediction rate [4, 49]. The data instances are from GCJ and the size of these data instances can be referred to Table 1. The results are reported in Figure 6.

It can be observed that our DSFM can achieve the lowest computation time compared to the baseline approaches, with 0.5×10^{-3}s for predicting one data instance. The prediction rate of our DSFM is 2,000 (i.e., 1 / (0.5×10^{-3})) data instances, which means that our DSFM can handle 2,000 instances in one second. Our DSFM is faster than 4 baselines, accelerating code2vec by 1.4×, code2seq by 3.5×, TBCCD by 12.0×, and ASTNN by 16.6×. As subtree-based neural networks, DSFM and ASTNN perform very differently with regard to efficiency. The reason is that DSFM adopts a more advanced acceleration method for subtree batch processing and the introduction of block-level similarity will not influence the computational complexity ($O(BN)$ v.s. $O(BNWD)$), as illustrated in Section 3.5. Another effective baseline, TBCCD, is also notablely slower than our DSFM, which takes 12× time for one prediction. Therefore, we can conclude that the proposed approach is also very competitive with regard to efficiency.

The time for training a DSFM model is related to the size of dataset and the training epochs. On GCJ, with 0.5 million code pairs for training, DSFM costs about 6.87 minutes per epoch and about 34.35 minutes for completing 5 epochs of training, indicating the practicality of DSFM.

4.4 RQ3: Analysis of Our Approach

To better understand our approach, we conduct experiments to show how does our DSFM perform with alternative settings.

4.4.1 Robustness Analysis. We first study the effectiveness of our DSFM when trained and tested on various datasets, aiming to evaluate its robustness. In this experiment, the training and test sets of each of the three datasets (i.e., GCJ, OJClone, and BigCloneBench) are coupled for model evaluation. As all datasets are not sufficiently large and have various data distributions, we additionally use 10% of the training set of the target dataset to fine-tune the trained model, and use the test set of the target dataset to evaluate the fine-tuned model. The results are reported in Table 4. It can be observed that our DSFM can still perform well on all combinations of training sets and test sets. Specifically, according to Table 2, it can be seen that DSFM achieves an F1 score of 97.3%, 99.6%, and 96.5% on GCJ, OJClone, and BigCloneBench, respectively. The performance of our DSFM that trained and tested on different sets can approximate those results, demonstrating the robustness of our DSFM.

Table 5: Ablation study using GCJ. "–" denotes the ablation of the corresponding component.

Method	Precision	Recall	F1 score
DSFM	99.4	95.3	97.3
– RvNN for subtree	53.4	66.9	59.4
– GRU for subtree sequence	89.1	59.7	71.5
– Program-level similarity	97.2	95.7	96.5
– Block-level similarity	96.0	93.0	94.5

4.4.2 Ablation Study. As shown in Table 5, we conduct an ablation study on components of DSFM to investigate their contribution to the performance. The first component to explore is the structural and sequential information introduced by RvNN and GRU, which mainly contribute to the quality of embeddings. We observe that removing RvNN suffers from a more serious decline in F1 score than removing GRU, dropping to 59.4% and 71.5%, respectively. This illustrates that the structural information of source code is much more important. But the sequential information introduced by GRU cannot be ignored either, which leads to a significant decline in F1 score of 25.8%.

We also ablate the program-level and block-level similarity to illustrate their efficacy. We note that the effect of removing scoring layer is slighter than removing RvNN or GRU. The reason are two-fold: 1) Feature extraction is an upstream process that can easily influence downstream processes. 2) Program-level and block-level similarity can both detect code clones independently. Another observation is that the block-level similarity is more important compared with the program-level similarity, which reflects in the fact that removing block-level similarity results in more significant performance degradation. Lastly, it is worth noting that DSFM without the block-level similarity still outperforms ASTNN (94.50% v.s. 93.78% in F1 score), demonstrating the effectiveness of the crafted block-level subtree construction adopted by DSFM.

4.4.3 Choices of Scoring Layer. Last investigation in our experiments is about the scoring layer. In our work, we implement the deep learning-based comparison paradigm by introducing deep subtree interactions. However, how to design the scoring layer to realize comparisons needs to be further studied.

We consider the following common first-order terms:

- $\oplus \rightarrow W^{1\times d}$: Add up all vectors and pass it to a fully-connected layer. It is a typical first-order term in FMs.
- $\max \rightarrow \odot \rightarrow 1^{1\times d}$: Apply element-wise max pooling to two groups of vectors separately, and operate inner product on two vectors. It is a famous distance metric.
- $\max \rightarrow \odot \rightarrow W^{1\times d}$: Apply element-wise max pooling to two groups of vectors separately, operate element-wise product on two vectors, and pass to a fully-connected layer. It is a common second-order term in FMs.
- $\max \rightarrow [] \rightarrow W^{1\times 2d}$: Apply element-wise max pooling to two groups of vectors separately, concatenate these two

Table 6: Results of DSFM using various scoring layers on GCJ.

Scoring Layer		**F1 score**
First-order (Program-level)	**Second-order (Block-level)**	
$\oplus \rightarrow W^{1\times d}$	$\odot \rightarrow 1^{1\times d}$	95.7
$\oplus \rightarrow W^{1\times d}$	$\odot \rightarrow W^{1\times d}$	95.9
$\oplus \rightarrow W^{1\times d}$	$[] \rightarrow W^{1\times 2d}$	46.6
$\max \rightarrow \odot \rightarrow 1^{1\times d}$	$\odot \rightarrow 1^{1\times d}$	91.5
$\max \rightarrow \odot \rightarrow 1^{1\times d}$	$\odot \rightarrow W^{1\times d}$	97.3
$\max \rightarrow \odot \rightarrow 1^{1\times d}$	$[] \rightarrow W^{1\times 2d}$	94.5
$\max \rightarrow \odot \rightarrow W^{1\times d}$	$\odot \rightarrow 1^{1\times d}$	95.4
$\max \rightarrow \odot \rightarrow W^{1\times d}$	$\odot \rightarrow W^{1\times d}$	95.3
$\max \rightarrow \odot \rightarrow W^{1\times d}$	$[] \rightarrow W^{1\times 2d}$	95.7
$\max \rightarrow [] \rightarrow W^{1\times 2d}$	$\odot \rightarrow 1^{1\times d}$	94.7
$\max \rightarrow [] \rightarrow W^{1\times 2d}$	$\odot \rightarrow W^{1\times d}$	97.1
$\max \rightarrow [] \rightarrow W^{1\times 2d}$	$[] \rightarrow W^{1\times 2d}$	24.1

vectors and pass to a fully-connected layer. It is a common distance metric.

We consider the following common second-order layers:

- $\odot \rightarrow 1^{1\times d}$: Perform inner product on every two vectors from different codes. It is a common distance metric.
- $\odot \rightarrow W^{1\times d}$: Perform element-wise product on every two vectors from different codes and pass to a fully-connected layer. It is a common second-order term in FMs.
- $[] \rightarrow W^{1\times 2d}$: Concatenate every two vectors from different codes and pass them to a fully-connected layer.

Table 6 shows the results of all possible combinations of first-order and second-order terms, where the highest 3 results are marked with gray color, from dark to light. We observe that the combination of the first-order term max $\rightarrow \odot \rightarrow 1^{1\times d}$ and the second-order term $\odot \rightarrow W^{1\times d}$ is the most effective, which is also chosen as the scoring layer in this work. Interestingly, some combinations perform very badly in F1 score. This is possibly because these combinations may cause fitting problems, making the variant hard to be well-trained. Besides, we note that the difference in performance between the highest 3 scoring layers is not significant. When facing different scenarios in real-world applications, either combinations can be considered for generalizability.

5 THREATS TO VALIDITY

Conclusion Validity. Some approaches with limitations are not introduced as baselines. For example, CDLH [46] is proven to be very time-consuming [50] and has not been made public by the authors. DeepSim [57] does not provide the detailed implementation of its feature matrix transformations. FCCA [20], SCDetector [48], HOLMES [31], and code2vec+graph2vec[52] require that the code snippets are compilable, whereas many cropped code segments are not completely and standardizedly described with lack of the dependency libraries for the source code files. The approach proposed in [14] involves caller-callee relations to analyze, yet the dataset such as BigCloneBench does not contain inter-procedural programs. CCGraph [58] is a PDG-based clone detector and was claimed by its authors to be hard to apply on unified datasets. Therefore, we could not replicate these approaches on our experimental settings such that do not introduce them as baselines.

In addition, there are no unified experimental setup in recent studies. For instance, the number of random data samples selected for dataset construction varies from 20 thousand [20], 50 thousand [53], 100 thousand [8], and 1 million [50]. The sampling method includes random sampling and balanced sampling. The evaluation on various types of clones is also different. DeepSim [57] trains its model on the overall dataset and distributes non-clone pairs to each clone type for evaluation. ASTNN [53] trains its model on the dataset of each clone types respectively and report the F1 score on the corresponding dataset. All the above different experimental setup may introduce evaluation bias to the competing approaches, which affects the experiment results to some extent.

Internal Validity. In our experiments, we report the highest performance achieved by clone detectors as their final results, aiming to measure their upper bound performance. Nevertheless, the stability of clone detectors to distinguish clones and non-clones is also of great importance. We should also consider measuring the average performance and its confidence interval as the results, which we leave for future work. Besides, DSFM depends on an AST parser to transform code snippets into ASTs. The widespread use of our DSFM requires suitable AST parsers for different programming languages. The AST parser can also have influence on the performance of our DSFM, which has not been explored sufficiently in our experiments. In the future, we will develop AST parsers for DSFM to adapt to a variety of programming languages. In this way, our approach can be more widely applied.

Construct Validity. Similar to the previous work [14, 31, 32, 38, 39, 42, 46, 48, 50, 53, 56, 57], the evaluation metrics used in this work are chosen as precision, recall, and F1 score. These metrics allow us to easily compare with the related work. In our future work, we will also consider other metrics for more comprehensive evaluation of the proposed approach.

External Validity. Two used datasets, GCJ and OJClone, are generated automatically based on functionalities that code snippets belong to. In addition, the source codes of GCJ and OJClone do not come from real-world production environments. The current experiment results only provide evidence that the proposed DSFM can well identify code clones that are syntactically dissimilar. Therefore, more investigation in real-world scenarios should be considered.

6 RELATED WORK

6.1 Code Representation Learning

Code representation learning is a technique that represents a source code snippet as a distributed vector, which contains abundant information about that source code. Allamanis et al. [2] extend ASTs to graphs with a variety of code dependency edges, and use Gated Graph Neural Network (GGNN) to represent code. Zhang et al. further improve this work and introduce heterogeneous graph neural network to learning representations [54]. code2vec [4] and code2seq [3] extract leaf-to-leaf paths from ASTs, and apply attention mechanism to code representations. ASTNN [53] is designed by partitioning ASTs into smaller flattened subtrees to learn code representations. By stipulating a robust distributional hypothesis for code, inst2vec [7] draws distributed code representations based on contextual flow. Bui et al. [8] propose a self-supervised learning paradigm by predicting subtrees. Unfortunately, all these approaches attach too much importance for projecting useful information to neural models, which hinders the great potential of subtree interactions for detecting functional code clones.

6.2 Code Clone Detection

Recently, code clone detection as one of the most important software engineering problems has attracted much attention. There are two kinds of pipelines to detect code clones. The first pipeline is based on the comparison. CCFinder [24] applies the transformation of source code and token-by-token comparison. Deckard [23] computes certain characteristic vectors to approximate structural information within ASTs and adopts locality-sensitive hashing to cluster these vectors. SourcererCC [39] is a token-based approach that detects code clones at the lexical level by comparing bag-of-tokens. CCAligner [44] exploits a code window that considers e-edit distance for matching to detect code clones. CCGraph [58] optimizes Program Dependency Graph (PDG)-based tools by using subgraph isomorphism and adopts the Weisfeiler-Lehman graph kernel to detect code clones. The second pipeline is recent deep learning-based approaches. White et al. [47] propose to use a recurrent and a recursive neural network to model the lexical and structural information to detect clones. CDLH [46] utilizes TreeLSTM [43] to embed ASTs and applies a hashing layer for functional code clone detection. TBCCD [50] adopts tree-based convolutions over ASTs to detect functional code clones. DeepSim [57] encodes data flow and control flow into a matrix and leverages a fully-connected layer to detect functional code clones. SCDetector [48] first conducts centrality analysis on the control flow graph and performs GRU on enhanced tokens. TECCD [15] conducts random walk and sentence2vec to generate code embeddings for detecting functional code clones. Fang et al. [14] and Yuan et al. [52] exploits semantic and syntactical information for detecting functional code clones. However, the aforementioned approaches either cannot handle more than Type-3 clones or pay much attention to the quality of code representation learning, leading to the detection methods being ineffective. In this work, we propose deep subtree interactions that compare the embeddings of every two subtrees of code snippets, aiming to improve the detection performance on Type-4 clones.

7 CONCLUSION

In this paper, we present an effective novel approach named DSFM based on a deep learning-based comparison paradigm. The proposed approach first divides the AST into a collection of subtrees via preorder traversal, and applies RvNN and GRU to produce the embeddings of those subtrees. To measure the functional similarity of two code snippets, we incorporate both the program-level similarity and block-level similarity. That is, we improve the typical deep learning-based approaches with deep subtree interactions, which compare every two subtrees between two code snippets. We have conducted extensive experiments on three widely-used datasets. The experimental results demonstrate that our DSFM is more effective and can outperform the state-of-the-art.

The replication package including source code and datasets is available at: https://github.com/xu-zhiwei/DSFM.

ACKNOWLEDGMENTS

This research is sponsored in part by the National Natural Science Foundation of China (No. 92267203, No. U20A6003, No. 62076146, No. 62021002, No. U19A2062, No. U1911401, No. 6212780016), the National Key Research and Development Program of China (No. 2020YFB1707700, No. 2022YFB3103903), and the Industrial Technology Infrastructure Public Service Platform Project "Public Service Platform for Urban Rail Transit Equipment Signal System Testing and Safety Evaluation" (No. 2022-233-225), Ministry of Industry and Information Technology of China.

REFERENCES

[1] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. 2017. Understanding of a convolutional neural network. In *Proceedings of the International Conference on Engineering and Technology (ICET)*. 1–6.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages (POPL)* 3 (2019), 1–29.

[5] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. 368–377. https://doi.org/10.1109/ICSM.1998.738528

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

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, Vol. 31.

[8] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-supervised learning of code representations by predicting subtrees. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 1186–1197.

[9] Geetica Chatley, Sandeep Kaur, and Bhavneesh Sohal. 2016. Software clone detection: A review. *International Journal of Control Theory and Applications* 9 (2016), 555–563.

[10] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. 2021. Understanding code reuse in smart contracts. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 470–479.

[11] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[12] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. 516–519.

[13] Yingnong Dang, Dongmei Zhang, Song Ge, Ray Huang, Chengyun Chu, and Tao Xie. 2017. Transferring code-clone detection and analysis to practice. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 53–62.

[14] 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 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 516–527.

[15] Yi Gao, Zan Wang, Shuang Liu, Lin Yang, Wei Sang, and Yuanfang Cai. 2019. TECCD: A tree embedding approach for code clone detection. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 145–156.

[16] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: Cross-project code reuse in github. In *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories (MSR)*. 291–301.

[17] Xiangnan He and Tat-Seng Chua. 2017. Neural factorization machines for sparse predictive analytics. In *Proceedings of the International ACM SIGIR conference on Research and Development in Information Retrieval (SIGIR)*. 355–364.

[18] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[19] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[20] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2021. FCCA: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability (TR)* 70, 1 (2021), 304–318.

[21] Ozan Irsoy and Claire Cardie. 2014. Deep recursive neural networks for compositionality in language. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, Vol. 27.

[22] Alon Jacovi, Oren Sar Shalom, and Yoav Goldberg. 2018. Understanding convolutional neural networks for text classification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

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

[24] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE)* 28, 7 (2002), 654–670.

[25] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 595–614.

[26] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[27] Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–38.

[28] Maggie Lei, Hao Li, Ji Li, Namrata Aundhkar, and Dae-Kyoo Kim. 2022. Deep learning application on code clone detection: A review of current knowledge. *Journal of Systems and Software* 184 (2022), 111141.

[29] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering (TSE)* 32, 3 (2006), 176–192.

[30] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[31] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. 2021. Modeling functional similarity in source code with graph-based siamese networks. *IEEE Transactions on Software Engineering (TSE)* (2021).

[32] 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 AAAI Conference on Artificial Intelligence (AAAI)*.

[33] Aravind Nair, Avijit Roy, and Karl Meinke. 2020. funcGNN: A graph neural network approach to program similarity. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.

[34] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "naturalness" of buggy code. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 428–439.

[35] Steffen Rendle. 2010. Factorization machines. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*. 995–1000.

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

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

[38] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 354–365.

[39] 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 International Conference on Software Engineering (ICSE)*. 1157–1168.

[40] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. CAST: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[41] Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. 2011. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 151–161.

[42] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the IEEE/ACM International Conference on Software Maintenance and Evolution*. 476–480.

[43] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing (ACL-IJCNLP)*.

[44] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. 2018. CCAligner: a token based large-gap clone detector. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1066–1077.

[45] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[46] 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 International Conference on Artificial Intelligence (IJCAI)*. 3034–3040.

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

[48] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: software functional clone detection based on semantic tokens analysis. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 821–833.

[49] Zhiwei Xu, Min Zhou, Xibin Zhao, Yang Chen, Xi Cheng, and Hongyu Zhang. 2023. xASTNN: Improved Code Representations for Industrial Practice. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[50] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the IEEE/ACM International Conference on Program Comprehension (ICPC)*. 70–80.

[51] Yijun Yu. 2019. fAST: flattening abstract syntax trees for efficiency. In *Proceedings of the IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 278–279.

[52] Dawei Yuan, Sen Fang, Tao Zhang, Zhou Xu, and Xiapu Luo. 2022. Java code clone detection by exploiting semantic and syntax information from intermediate code-based graph. *IEEE Transactions on Reliability (TR)* (2022).

[53] 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 IEEE/ACM International Conference on Software Engineering (ICSE)*. 783–794.

[54] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to represent programs with heterogeneous graphs. In *Proceedings of the IEEE/ACM International Conference on Program Comprehension (ICPC)*. 378–389.

[55] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–38.

[56] Yan-Ya Zhang and Ming Li. 2019. Find me if you can: Deep software clone detection by exploiting the contest between the plagiarist and the detector. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 33. 5813–5820.

[57] Gang Zhao and Jeff Huang. 2018. DeepSim: deep learning code functional similarity. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 141–151.

[58] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 931–942.