From Local to Global Semantic Clone Detection

1st Given Name Surname
department name of organization (of Aff.)
name of organization (of Aff.)
City, Country
e-mail address or ORCID

2nd Given Name Surname
department name of organization (of Aff.)
name of organization (of Aff.)
City, Country
e-mail address or ORCID

3rd Given Name Surname
department name of organization (of Aff.)
name of organization (of Aff.)
City, Country
e-mail address or ORCID

4th Given Name Surname
department name of organization (of Aff.)
name of organization (of Aff.)
City, Country
e-mail address or ORCID

5th Given Name Surname
department name of organization (of Aff.)
name of organization (of Aff.)
City, Country
e-mail address or ORCID

6th Given Name Surname
department name of organization (of Aff.)
name of organization (of Aff.)
City, Country
e-mail address or ORCID

Abstract—Clone detection detects similar code fragments (refer to as clone code) in software products. It can help with software optimization and maintenance. Code clone detection can be divided into textual, lexical, syntactic and semantic levels. The existing technologies have achieved many good results in the first three levels, but no significant results have been obtained in semantic clone detection. In this paper, we propose a novel semantic level clone detection approach. We use the control flow graph (CFG) as an intermediate representation of the program method, combining the classical dynamic time warping (DTW) algorithm in the field of speech recognition with two deep neural network models (bidirectional RNN autoencoder and graph convolutional network (GCN)) to detect semantic level clone from local to global. We experimented with a dataset consisting of five large-scale real-world systems and a code corpus containing a large number of programming problems. The experimental results show that our approach can achieve good results in detecting both local and global semantic clone.

Index Terms—clone detection, semantic clone, deep learning, dynamic time warping

I. INTRODUCTION

Duplicate or similar code often appears in software systems, referred to as clone code. Clone detection focuses on finding these clone code fragments. Most of the current clone detection techniques are used to reduce the negative impact of the clone. For example, plagiarism detection uses clone detection techniques to detect copyright infringements between large scale software products [1] [2]. Besides, unnecessary clone increases the cost of software maintenance. For example, once a bug is found in a code fragment, we have to reevaluate all the code that was copy-paste-modified by it. To solve this problem, clone detection can be used to alert programmers to merge clone code during development and to locate clone code during maintenance [3]. In addition, software quality assessment can be performed by detecting the proportion of clone code [4], and software evolution analysis can be performed by analyzing the trend of clone code between different software versions [5]. By detecting clone code between malware, we can study their behavior patterns and use these patterns to detect malware [6]. By building a defect knowledge base, when a software failure occurs, it can automatically provide repair suggestions by detecting defect code similar to the fault code in it. In a word, code clone is not always harmful [7] [8], but clone detection is a critical issue for many applications.

Roy et al. define code clones as four types [9]. Type-1: Code fragments that differ only in white space, layout, and comments. Type-2: In addition to spaces, layouts, and comments, there are different identifiers, literals, and data types. Type-3: In addition to the above, some modifications such as changing, adding or deleting code statements have been added. Type-4: Two or more code fragments that perform the same calculation but are implemented by different syntactic variants. Existing clone detection techniques and tools have achieved many good results in detecting textual, lexical and syntactic clones, corresponding to Type-1, Type-2, and Type-3, respectively [10] [11] [12]. However, since the Type-4 clone is more difficult to detect, no significant results have been achieved. Type-4 clone detection, also known as semantic clone detection, is used to detect the similarity of programs in functional (rather than textual), and researchers have realized its necessity. On the one hand, clone types are inclusive, clone code fragments from textual to syntactic level are likely to be semantically equivalent. On the other hand, when comparing functional similarities, the input is usually the abstract program skeleton. Therefore, the semantic clone detection approaches are suitable for detecting cross-language programs [13], and even for informal program description languages, such as algorithm pseudocode [14].

In this paper, we further subdivide the semantic level clones into two types: local semantic clone and global semantic clone. Local semantic clone refers to one or more lines that are functionally equivalent. We give some examples of local semantic clone in Table I. Conversely, global semantic clone refers to program methods that differ in overall structure or even implementation logic but are functionally identical, e.g. the recursive and iterative implementation of the Fibonacci sequence (as shown in Fig. 1). According to this division, we propose a novel approach to detect semantic clone of these two...
TABLE I
THREE EXAMPLES OF LOCAL SEMANTIC CLONE

<table>
<thead>
<tr>
<th>Local Semantic Clones</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program statements perform the same function</td>
<td>for / while / do...while</td>
</tr>
<tr>
<td>Hoist expressions play the same judgment</td>
<td>i &lt;= 0 / !(i &gt; 0)</td>
</tr>
<tr>
<td>Step-by-step and comprehensive expression</td>
<td>s = a * a + 1; / s = a * a ; ++s;</td>
</tr>
</tbody>
</table>

public static int Fibonacci1(int n) {
    if (n == 0 || n == 1)
        return 0;
    return Fibonacci1(n - 1) + Fibonacci1(n - 2);
}

public static int Fibonacci2(int n) {
    int first = 0, second = 1, result = 0;
    if (n == 0 || n == 1)
        return n;
    for (int i = 2; i <= n; ++i) {
        result = first + second;
        first = second;
        second = result;
    }
    return result;
}

more refined types. The comparison units of our approach are program methods, we use the control flow graph (CFG) as their intermediate representation. We use deep learning techniques to construct two deep neural network models, a node feature model, and a CFG feature model. The node feature model models the nodes in the CFG and then leverages the traditional dynamic programming (DP) idea to calculate the possibility of a method pair being a local semantic clone pair. The CFG feature model models the CFG itself, which is used to detect global semantic clone. More specifically, the node feature model is a bidirectional recurrent neural network (RNN) autoencoder [15] [16] [17] [18] [19] [20], and the CFG feature model is a graph convolutional network (GCN) [21] [22] [23]. The DP algorithm we use is the dynamic time warping (DTW) [24] algorithm.

There are already some clone detection techniques combined with deep learning. For example, White et al. [25] use RNN to learn lexical level features of the code and then use the recursive neural network to learn syntactic level features. They use the Abstract Syntax Tree (AST) as an intermediate representation, and the highest level clone that can be detected is the syntactic level. Li et al. [26] use the token sequence of the code as an intermediate representation and then extract the frequency of each token category. These frequencies are used as input data to train the neural network to distinguish between clone and nonclone pairs. Their approach is also difficult to detect semantic level clone. In fact, since semantic clone pairs are not necessarily syntactically similar, the results obtained by token-based and tree-based methods are always less than satisfactory [27] [28] [29]. Some traditional non-deep learning methods obtain semantic clone pairs by detecting isomorphic subgraphs in the program dependence graph (PDG) [30] [31], but the final recall rates are rarely more than 0.5. Yu et al. [32] attempted to detect semantic clone using deep learning convolution techniques on AST with a final recall rate between 0.41 and 0.62, which is still a tree-based approach. Therefore, we believe that if the convolution operation can be applied to the graph structure representation of the code, better results can be obtained. In addition, we are not the first to use the DTW algorithm for clone detection, Abdelkader et al. [33] have used a similar approach. However, they simply block the code by statement and use the ASCII of all the characters in each block as a time series. The distance is considered to be 0 when the corresponding ASCII codes in the two statements are the same, and 1 otherwise. Their research only proved that treating code as a time series can detect a certain degree of clones. In addition, their method can only detect clones of Type-1 to Type-3. We introduced deep learning to map source code more accurately into time series, greatly improve the precision of the result. Besides, since the points in the time series are nodes in the CFGs, our approach can detect semantic level clones.

We use five real-world open-source systems to validate our local semantic clone detection approach and build a global semantic clone dataset to validate our global semantic clone approach. For local semantic clone, by manually judging the correctness of the test results, we find that our approach can not only detect local semantic clone but also almost make no false positives. For global semantic clone, experiment results show that when the ratio of the clone pairs to the nonclone pairs in a dataset is between 1:1 and 1:20, the recall rate can reach 0.62 to 0.95.

The contributions of this paper are as follows:

- We propose the idea of subdividing semantic clone into local semantic clone and global semantic clone. Through such a division, it is easy to explore the detection methods based on the different characteristics of the two types.
- We propose an approach to combine the traditional algorithm with deep learning techniques that can be directly applied to CFG. Through our approach, both local and global semantic clones can be effectively detected.

The rest of this paper is organized as follows: In Section II, we introduced the neural network architectures we used. In Section III, we describe our approach in detail. In Section IV, we describe our experiments and results. Finally, we conclude in Section V and provide some suggestions for future work.

II. PRELIMINARIES

A. Autoencoder

Autoencoder [15] [16] is an unsupervised learning algorithm. It is a three or more layer neural network that attempts to approximate the output as a copy of the input. As shown in Fig. 2, the goal is to learn the parameters \( W \) and \( b \) such that \( f_{W,b}(x) \approx x \). In order to make \( f \) meaningful, the number of neurons in hidden layer \( h \) is limited. Depending on whether the neurons in \( h \) are more or less than the input layer \( x \), they can...
be considered as compressed or sparse representation vectors for the input data.

B. RNN

RNN [17] [18] is a neural network for processing sequence data. As shown in Fig. 3, the forward RNN scans the input data from left to right. \( x_t \) and \( a_t \) are the input and activation values of time step \( t \), respectively. Each \( a_t \) is a function of \( a_{t-1} \) and \( x_t \). Therefore, when the output of the time step \( t \) is predicted, not only the input information \( x_t \) of the current step but also the information of the time steps \( x_1, x_2, \cdots, x_{t-1} \) can be obtained. We can construct both a forward RNN and a backward RNN at the same time so that at any point in the sequence, we can get information from all time steps before and after it. This type of RNN model is called a bidirectional RNN [19].

C. GCN

GCN [21] [22] is a neural network architecture proposed in recent years. The convolutional neural network (CNN) in deep learning is designed to process neatly arranged Euclidean structural data, such as one-dimensional speech data or two-dimensional digital image data. GCN extends it to graph data with arbitrary structure [23]. By stacking graph convolutional layers, the GCN can learn the proportion of information that each node should extract from its neighbors in order to complete a classification task. This is equivalent to the structural features of the graph.

The most commonly used graph convolution formula is (1), where \( \tilde{A} = A + I_N \), \( A \) is the adjacency matrix, \( \tilde{D}_{ii} = \sum_j \tilde{A}_{ij} \), \( X \in \mathbb{R}^{N \times F_1} \) is a node feature matrix composed of feature vectors of each node. This formula transforms the feature space of the nodes from the space where \( X \) is located to space where \( Z \) is located, and \( \Theta \in \mathbb{R}^{F_1 \times F_2} \) is the parameter to learn. \( N \) is the number of nodes of the graph, \( F_1 \) and \( F_2 \) are the dimensions of the node feature vectors in the original and new feature spaces, respectively [22].

$$ Z = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta $$

(1)

Here we give a simple example to qualitatively understand this formula. The left side of Fig. 4 is a graph with 4 nodes, each with a self-loop (shallow circle in the figure). It can be observed that the product of \( \tilde{A} \) and \( X \) is exactly the vector sum of each node and its neighbors. That is, \( \tilde{A} \cdot X \) provides each node with information about all its neighbors, which is combined with the node’s own information to update the node.

The degree matrix \( \tilde{D} \) is used for normalization. Thus, \( \Theta \) can be thought of as the proportion of information that should be extracted from each node.

III. APPROACH

In this section, we first outline our approach in Section III-A. After that, we detailed three key steps.

A. Overview

Our approach is divided into two phases: the training phase and the detecting phase. The steps for each phase are listed below, and the overall flow is shown in Fig. 5.

The training phase consists of the following three steps:

- **Step 1**: Generate a CFG for each method in the training set. Get the adjacency matrix and node text labels. These text labels form a text label corpus (Section III-B).
- **Step 2**: Construct a bidirectional RNN autoencoder and train it using the text label corpus to obtain the node feature model (Section III-B).
- **Step 3**: Construct a GCN, take the node feature matrix and the adjacency matrix of the CFGs as inputs, train to obtain the CFG feature model (Section III-D).

The detecting phase consists of the following four steps:

- **Step 1**: Generate a CFG for each method in the code set to be detected, get its adjacency matrix and node text labels (Section III-B).
- **Step 2**: Generate a node feature matrix for each CFG using the node feature model (Section III-B).
- **Step 3**: Calculate the distance between each pair of node feature matrices using the DTW algorithm to obtain local semantic clones (Section III-C).
- **Step 4**: Generate a CFG feature vector for each CFG using the CFG feature model. Calculate the cosine similarity between each pair of these vectors to obtain global semantic clones (Section III-D).

B. Extract Information from CFGs

We use the Java analysis tool Soot [34] [35] to generate CFGs that are used as input for our detection. It’s worth noting that although we mentioned Soot here, our approach is not limited to Java, we just take advantage of some of the natural advantages of Soot. It introduces a lot of rules when generating CFGs to unify the fragments we call local clones. For example, the two program methods on the left side of Fig. 6 contain the three local clones we listed in Table I. However, despite the large differences in the surface between the two methods, the
CFGs that Soot generates for them are identical. As we can see, Soot represents each atomic operation in the program as a CFG node and generates a text label for each node. The pointing relationship between nodes represents the order in which programs are executed. In the text label, the variable name is uniformed, but information such as variable values or string literals is retained. Variable names are abstracted by a combination of “letter + number”. The letter indicates the type of the variable (e.g. “i” for int, “d” for double), the number indicates the entry number, and the variables starting with “$” are the temporary variables.

The next thing to do is that for each text label in the CFG, we generate a representation vector, i.e. the node feature vector. We construct a node feature model to obtain these feature vectors. We introduced the autoencoder and the bidirectional RNN in Section II. We use a bidirectional RNN autoencoder [20] to build the node feature model. The model is divided into three parts: the encoder, the decoder, and the replicator. Fig. 7 is an illustration of the model. In the encoder, any text label with n characters is treated as a sequence of $T_x = n$ time steps. In each time step $t$, the one-hot coding $o_{c_t}$ of the character $c_t$ is taken as input. We provide these one-hot coding to the forward and backward RNNs. The output $h$ of the forward RNN of the last step and the output $\overrightarrow{h}$ of the backward RNN of the first time step are stacked together into a vector $h$. In all $T_y = T_x$ time steps, the vector is copied $T_x$ times as input to the decoder. The decoder is also a bidirectional RNN. Then, we stack the forward and backward outputs of the decoder at each time step to get $y_1, y_2, \ldots, y_{T_y}$.

Our goal is to iteratively update the parameters in the model so that the difference between each $y_t$ and $z_t$ is getting smaller and smaller. After training, $h \in \mathbb{R}^F$ in the replicator is the node feature vector. All node feature vectors in the CFG are combined to form a node feature matrix of the CFG. Finally, we obtain the adjacency matrix $A \in \mathbb{R}^{N \times N}$ directly from the
C. Local Semantic Clone Detection

After converting the local clone fragments in the source code to the same subgraphs, we can use CFGs to directly calculate the similarity of two methods. Specifically, we can calculate the similarity between their node feature matrices. However, the numbers of nodes in the CFGs of different methods are different, and the difference in the numbers of nodes results in different dimensions of the node feature matrices. In addition to this, the same code fragments in both methods may appear in different locations due to add or remove statements, which makes the corresponding vectors in the node feature matrices also appear in staggered positions. We give a simple example in Fig. 8, func2 in part A adds an “if divide by 0” statement based on func1. This corresponds to the fact that the CFG for func2 has four more nodes \( (n_2, n_3, n_4, n_5) \) than the CFG for func1 (in part B). After feeding the two CFGs into the node feature model, the node feature matrices of two different dimensions are obtained, and the feature vectors corresponding to the same code fragment Fragment 2 are completely staggered therein (in part C).

It is easy to understand that the general matrix similarity measures cannot calculate the similarity between the node feature matrices of different CFGs. To solve this problem, we use the DTW algorithm. DTW is a classic algorithm for speech recognition of isolated words. It is used to solve the difficulty of recognising caused by different speech speeds, intonations or breath sounds when different speakers speak the same word. As part D in Fig. 8, the DTW algorithm automatically finds the matching point in the time series, just like its name, it can “warp” the time. We apply this algorithm to the similarity calculation of the node feature matrices.

For any pair of CFGs \( G_1 \) and \( G_2 \) (corresponding to program methods \( m_1 \) and \( m_2 \)) with \( N_1 \) and \( N_2 \) nodes, respectively. According to the method in Section II-B, we obtain their node feature matrices with dimensions \( N_1 \times F \) and \( N_2 \times F \), respectively. Then, we create a matrix \( D \) of shape \( N_1 \times N_2 \). The element \( d_{i,j} \) in \( D \) is the cosine distance of the feature vectors of the node \( i \) in \( G_1 \) and the node \( j \) in \( G_2 \). The cosine distance is obtained by “1 - cosine similarity”, we treat all cosine similarities with negative values as 0, so the range of element values in \( D \) is 0 to 1. Then, we take matrix \( D \) as input and get the DTW distance between \( G_1 \) and \( G_2 \). Specifically, a new matrix \( \Delta_{DTW} \) is defined to store the cumulative DTW distances. For each element \( \delta_{i,j} \) in \( \Delta_{DTW} \), to find the minimum cumulative DTW distance from the starting point to it, we use the minimum of \( \delta_{i-1,j}, \delta_{i,j-1}, \delta_{i,j-1} \) plus \( d_{i,j} \) in \( D \), as shown in (2) and (3). Finally, we get a DTW path. We use the normalized cumulative DTW distance at the endpoint (divided by the path length \( K \)) as the distance between the node feature matrices, i.e. the distance between \( G_1 \) and \( G_2 \), which is also the distance between \( m_1 \) and \( m_2 \).

\[
\delta_{i,j} = d_{i,j} + \min\{\delta_{i-1,j}, \delta_{i,j-1}, \delta_{i-1,j-1}\}
\]

\[
distance_{m_1, m_2} = \frac{1}{K} \delta_{N_1, N_2}
\]

The E part of Fig. 8 is an example. The number in the lower right corner of each element in the matrix represents the cosine distance \( d_{i,j} \) of the corresponding node feature vector, and the number in the upper left corner is the cumulative DTW distances \( \delta_{i,j} \). The DTW algorithm will automatically find the shortest DTW path from the start point to the endpoint. The \( \delta \) of the last element (1.84 in the figure) divided by the length of the shortest DTW path (9 in the figure) is the distance between the CFGs, which we consider as the semantic distance between \( func1 \) and \( func2 \).

We obtain the DTW distance of each method pair in this way. The smaller the distance, the higher the local semantic similarity between the methods. We can simply set a threshold \( t \) and think that all method pairs \( m_a \) and \( m_b \) corresponding to \( distance_{m_a, m_b} < t \) are local semantic clone pairs. We can also use the distances as input to a clustering algorithm (such as K-means or AP clustering) to obtain local semantic clone classes in the system.

It is worth noting that our approach not only recognizes local semantic clones but also recognizes the clones of Type-1 to Type-3 because all of these types of clones’ CFGs are the same in each clone pair.

D. Global Semantic Clone Detection

Unlike local semantic clone, the global semantic clone cannot be unified into the same CFG structure by simple transformation rules. We use the GCN in deep learning to construct a CFG feature model to learn these global semantic clones. The second role of the node feature matrices is as input to the CFG feature model. According to the graph convolution theory described in Section II-C, we use (1) to map all \( N \) nodes in a CFG from the original feature space to a new feature space. Equation (1) is called a graph convolutional layer. We use a Rectified Linear Unit (ReLU) to perform nonlinear activation on it. A graph convolutional network is obtained by stacking several nonlinearly activated graph convolutional layers. More specifically, we use multi-layer attention based GCN inspired by [36]. We represent it by (4) and (5).

\[
H^{(l)} = \sum_{k=1}^{l-1} H^{(k)} \alpha^{(l)} + \text{ReLU} \left( \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l-1)} \phi^{(l)} \right) \cdot \alpha^{(l)}
\]

\[
\alpha_k^{(l)} = \frac{e_k^{(l)}}{\sum_{j=1}^{l} e_j^{(l)}}
\]

The letters in the brackets in the upper right corner of (4) and (5) represent the layers. \( \alpha \in \mathbb{R}^{F \times F} \) represents the attention matrix. For any \( l \)th layer, there are \( l \) attention matrices: from \( \alpha_1^{(l)} \) to \( \alpha_l^{(l)} \). They indicate that to obtain the output \( H^{(l)} \) of the \( l \)th layer, how much attention needs to be paid on the \( 1 \)st to \( l \)th layers. Equation (5) guarantees that the sum of all \( \alpha \) in the same layer is one (all-ones matrix). \( \phi^{(l)} \in \mathbb{R}^{(F \times F)} \) are parameters that need to be learned. The output \( H^{(L)} \) of the last nonlinear graph convolutional layer of the model is also an \( N \times F \) matrix. (We do not change the
dimensions of the node feature vector during the convolution operations.) We take the dimension reduction of $H^{(L)}$ by taking out the maximum values in the first dimension to get a vector of $F$ dimensions. After the L2-normalization of this vector, the output of the CFG feature model is obtained.

During the training phase of this model, we randomly select method pairs $m_i$ and $m_j$ from the dataset continually. Feed the node feature matrix and adjacency matrix of their CFGs into the model and get two output vectors, denoted by $\theta_i$ and $\theta_j$. We calculate the cosine similarity $\cos(\theta_i, \theta_j)$ of the two output vectors and compare it with the actual label $y_{i,j}$. We use the loss function in (6). In order to make the loss value smaller and smaller, when $m_i$ and $m_j$ are functionally identical program methods ($y_{i,j} = 1$), we want $\cos(\theta_i, \theta_j)$ to be as large as possible, when $m_i$ and $m_j$ are not functionally identical ($y_{i,j} = 0$), we want $\cos(\theta_i, \theta_j)$ to be as small as possible.

$$\text{Loss} = \sum_{i \neq j} y_{i,j} \cdot (1 - \cos(\theta_i, \theta_j))^2 + (1 - y_{i,j}) \cdot \cos^2(\theta_i, \theta_j) \quad (6)$$

In the detecting phase, when we want to determine whether a method pair is a global semantic clone pair, we only need to provide their CFGs to the model and compare the cosine similarity of the two output vectors. When detection is performed on a system, similar to Section III-C, we can calculate the cosine similarity of all method pairs in the system, get results directly by comparing them with a certain threshold. We can also use the cosine distance matrix as input to some clustering algorithms to obtain the global semantic clone classes.

IV. EXPERIMENTAL

This section is experimental verification and results. Section IV-A is an introduction to the dataset. In Section IV-B, we use a series of experiments to verify the feasibility and effectiveness of our approach for detecting clones (including syntactic and semantics). In Sections IV-C, we experimented to verify whether our approach can detect semantic level clones (both local and global). Finally, we do a detailed discussion in Section IV-D.

A. Dataset

Our dataset consists of two parts. The first part comes from five real-world large-scale Java systems. There are approximately 630,000 lines of code and 30,000 methods (including class member methods and constructors). The details of the dataset are shown in Table II. The second part of the dataset...
comes from the Codewars (https://www.codewars.com/) website. It is an online programming website with a lot of programming problems. Each problem has a number of user solutions, the functions of these solutions are the same, and these solutions can pass all the test cases of the problem, so we can guarantee their correctness. We chose 574 programming problems and roughly delete Type-1 to Type-3 clones, leaving only Type-4 clone (i.e. let all the solutions to each problem be as different as possible in textual), and ended up with an average of about 22 solutions per problem.

B. Whether the Approach Can Detect Clone?

As we mentioned earlier, the relationship between clone types is included, that is, higher-level clone detection results contain low-level clone pairs. Therefore, in this section, we will first verify the effectiveness of our approach for detecting all types of clones. Corresponds to the three key steps in Section III, our experiment wants to answer the following three research questions:

1) RQ 1: Are the node feature vectors and CFG feature vectors calculated by our node feature model and CFG feature model meaningful?

2) RQ 2: Whether the DTW distance between node feature matrices can be used to distinguish between clone and nonclone pairs?

3) RQ 3: Whether the cosine similarity between CFG feature vectors can be used to distinguish between clone and nonclone pairs?

1) RQ 1: Are the node feature vectors and CFG feature vectors calculated by our node feature model and CFG feature model meaningful? To answer this question, we visualize the feature vectors obtained by the two models. In our experiments, the two types of feature vectors are all 128-dimensional ($F = 128$ in Section III), and we use the t-SNE algorithm to reduce them to 2D and 3D space. This can be done very easily using Google’s open-source high-dimensional data interactive display tool Embedding Projector (https://projector.tensorflow.org/). We found that the vectors generated by each model formed many clusters. By observing the points in each cluster, we found that most of the points in the same cluster belong to the same category. In the node feature space, the points in each cluster correspond to the same or similar node text labels. In the CFG feature space, most of the CFG feature vectors corresponding to the same programming problem are also mapped to adjacent regions. This answers RQ 1.

2) RQ 2: Whether the DTW distance between node feature matrices can be used to distinguish between clone and nonclone pairs: To answer this question, we first introduce the clone detection tool Nicad [10]. Nicad is one of the most recognized clone detection tools. It focuses on near-miss clone due to intentionally copied fragments that may be modified to accommodate new contexts. Some comparative studies have shown that it exceeds or is equivalent to many other tools in recall and precision, especially for detecting syntactic clones. We use the latest version of Nicad (version 5.2, http://www.txl.ca/txl-nicaddownload.html), which was released in July 2019.

For the five datasets in Table II, we use Nicad to detect the clone pairs and then used our approach to calculate the DTW distances for all method pairs (throw away methods that are less than 10 lines as Nicad does). We plot the DTW distance kernel density estimation curves for all method pairs (both clone and nonclone) and pure clone method pairs (the
The kernel density estimation curve is a useful tool for observing the shape of the distribution. The height of the vertical axis can represent the probability density of points on the horizontal axis, area under the curve sum to 1. The numbers on the legend in Fig. 9. are the total number of method pairs for the corresponding category.

As can be seen from the figure, for all method pairs, the DTW distance values are approximately a normal distribution between 0 and 1 (the peaks are more to 1). At the same time, the DTW distance values of the pure clone method pairs are highly concentrated at positions close to zero. Even more gratifying, the coincident areas of the two curves are quite small. For a more precise explanation, we calculated the mean and median values of the DTW distances for all method pairs and pure clone method pairs, as shown in the upper rows of Table III. Among them, we can see that the average and median DTW distance of pure clone method pairs calculated by our approach are very small, all-around 0.05. In contrast, the mean and median of all method pairs including clone and nonclone was around 0.60. When we assume that Nicad’s detection results are all true clone pairs (Nicad does have high precision), this indicates that the DTW distance values obtained by our approach can effectively distinguish between clone pairs and nonclone pairs. This answers RQ 2.

3) **RQ 3:** Whether the cosine similarity between CFG feature vectors can be used to distinguish between clone and nonclone pairs: We randomly extracted 2000 methods from the Codewars corpus as the test set and the rest as the training set. The number of clone pairs and nonclone pairs contained therein is listed in Table IV. During the training process, we used the upsampling method due to the imbalance between clone and nonclone pairs (i.e. instead of always sampling from the entire training set, each time we pick five samples from the training set, we randomly pick a sample from the set of pure clone method pairs). To illustrate that semantic level clone can be identified by calculating the cosine similarity of the feature vectors generated by the CFG feature model, we calculated the mean and median of the cosine similarity for all semantic level clone and nonclone pairs, as shown in the last row of Table III. It is easy to see that the mean and median of the cosine similarity of the clone pairs are much larger than those of the nonclone pairs. Therefore, we believe that the CFG feature model can extract the information in the CFGs and generate meaningful coding vectors, and by comparing these vectors, we can distinguish between clone and nonclone pairs. In fact, since we have roughly cleaned up the dataset (so that the solution for each problem is as different as possible in textual), to some extent, this result has validated the effectiveness of our approach in detecting global semantic clones.

**C. Whether the Approach Can Detect Semantic Clone?**

We evaluated the result of local semantic clone detection on the five real-world Java systems. For these systems, we do not know how many clone pairs there are, and there are no ready-made labels to determine whether the clone pairs we detected are true clones or false, so we use a manual method. We detected the five Java systems and set the DTW distance threshold to 0.1, which means that all method pairs with DTW distance less than 0.1 are considered as clone pairs. From the result pairs, we removed the clones that were simultaneously discovered by Nicad (indicating that they are Type-1 to Type-3 clones), and then manually verified the rest. The results are shown in Table V. As can be seen from the table, almost all of the local semantic clone pairs detected by our approach are true clones, which shows that our approach has high precision. For those false clones, we found that these
TABLE V
LOCAL SEMANTIC CLONE DETECTION RESULTS (DTW DISTANCE THRESHOLD: 0.1, NUMBER OF LINES: 10-2500)

<table>
<thead>
<tr>
<th>Systems</th>
<th>Method</th>
<th>LOC</th>
<th>Time Cost (s)</th>
<th>Number of Clone Pairs Detected by</th>
<th>True Semantic Clone Pairs</th>
<th>False Semantic Clone Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Our approach</td>
<td>Nicad</td>
<td>Our approach only</td>
</tr>
<tr>
<td>Apache commons imaging</td>
<td>193016</td>
<td>537</td>
<td>548</td>
<td>179</td>
<td>109</td>
<td>10</td>
</tr>
<tr>
<td>Colt</td>
<td>26395</td>
<td>824</td>
<td>1142</td>
<td>996</td>
<td>846</td>
<td>150</td>
</tr>
<tr>
<td>Catalano</td>
<td>42637</td>
<td>1080</td>
<td>3491</td>
<td>504</td>
<td>459</td>
<td>45</td>
</tr>
<tr>
<td>Weka (without GUI)</td>
<td>53820</td>
<td>1439</td>
<td>3754</td>
<td>240</td>
<td>185</td>
<td>55</td>
</tr>
<tr>
<td>Apache commons math3</td>
<td>52737</td>
<td>1521</td>
<td>5112</td>
<td>309</td>
<td>358</td>
<td>151</td>
</tr>
</tbody>
</table>

Fig. 11. Examples of Semantic Clone Pairs Detected by Our Approach.

clone pairs all contain a `switch` statement and a lot of `cases`. This causes the methods' codes to be structurally similar, which should be the reason of the detection error.

Although we have manually verified the possible clone pairs detected, in fact, we can explain the existence of similar code simply by method signatures. In Fig. 11, we list the package names, file names, and method signatures of the clone pairs we detected for the Catalano Framework, we highlighted the differences between each clone pair. Based on these method names, it is easy to understand that the functions they implement should be the same. We show the code for one of the local semantic clone pairs we detected in Fig. 12.

For the global semantic clone, we have simply demonstrated the feasibility of our approach in RQ 3. According to [37] and [38], the proportion of clone code in large-scale software systems in the real world is 7% to 23%. Therefore, we randomly sampled from the test set of the Codewars corpus and built several validation sets containing different ratios of clone and nonclone pairs (from 1:1 to 1:20). We use (7) to calculate the F1-score of the detection results of these verification sets. We select the largest F1-score among the detection results of each verification set, the corresponding recall, precision and cosine similarity threshold are shown in Table VI. We also show the trend of recall and precision at different clone ratios in Fig. 13.

\[
F_1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}
\] (7)

It can be observed that the higher the proportion of cloned code in the verification set, the better our approach behaves. This is because we used a 1:5 upsampling method during the training phase (five nonclone pairs followed by one clone pair). This shows that our idea of constructing a CFG feature model to detect global semantic clone is correct, but there is still much room for improvement in the training of the model.
D. Discussion

1) Internal Validity: The method we used to answer RQ 1 is a commonly used visualization method that simply evaluates clustering or classification results. It corresponds exactly to the nature of our two models. The node feature model clusters the same type of CFG nodes, and the CFG feature model classifies clone method pairs versus nonclone method pairs.

As for the comparison with Nicad, we have adopted the same configuration. The Nicad configuration we use is “type 3-2: threshold=0.3, rename=blind”, which means that the variable name and the entry number of the variable are not taken care of, and the method pairs with similarity exceeding 70% are considered clone pairs. Soot helps us unify the variable names, at the same time, we removed the numbers in the “letter + number” combination when extracting text labels.

2) External Validity: Deep neural network models are prone to over-fitting problems when the models are complex and the training data is insufficient. That is, the output of the training set can be predicted perfectly, but it cannot be generalized to other datasets. While training the node feature model, we found that the number of nonrepeating text labels rose more and more slowly as text labels were extracted from CFGs. We think this means that the dataset used to train the node feature model is sufficient. However, for the CFG feature model, we acknowledge the possibility of overfitting. We have taken some ways to avoid this problem. For example, we did not use a complex model or a large number of neurons, and we take a portion from the dataset for verification. Besides, because programmers from all over the world have various programming habits, codes randomly selected from Codewars can roughly form a uniform distribution.

3) Why this approach can detect 4 types of clones: We think it can be explained for many reasons, and we list some possible reasons below.

   - The text label generated by Soot contains the values of the variables and the string literals, which is equivalent to removing the unimportant text information and retaining the key text.
   - Soot unifies variable names by type. Also, it generates each atomic operation (not just the lines of codes) in the program as a node, so it is not affected by whitespace, comments, and variable names.
   - The node feature model can learn the features in the text label, the more similar the labels, the closer the relative positions in the feature space. For example, as we found in RQ 1, for statements that assign a string constant to a string variable, the labels generated by soot is $r = "..."$ or $r = "..."$. The nodes corresponding to these labels all fall in the same area in space. Not only that, the closer the string constants, the closer the spatial positions of the two nodes.
   - The DTW algorithm, like its name, can “warp” time, and the newly added or deleted statements in the cloned code are like those points that cannot be aligned in time series. Using the DTW algorithm is equivalent to skipping segments that cannot be aligned and automatically locating aligned segments. Therefore, the difference in small fragments does not affect the matching result of the entire sequence.
   - The GCN can extract structural features from the graph. Through our training, the CFG feature model can gradually find that two or more different structures can always achieve the same function, such as the structure corresponding to recursion and iteration. Detecting semantic clone is just to find these different structures that can complete the same task.

<table>
<thead>
<tr>
<th>Ratio</th>
<th>F1-score</th>
<th>Recall</th>
<th>Precision</th>
<th>Cosine Similarity Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1</td>
<td>0.9105</td>
<td>0.9398</td>
<td>0.8671</td>
<td>0.2</td>
</tr>
<tr>
<td>1:5</td>
<td>0.7671</td>
<td>0.9119</td>
<td>0.6621</td>
<td>0.4</td>
</tr>
<tr>
<td>1:10</td>
<td>0.6986</td>
<td>0.7316</td>
<td>0.6085</td>
<td>0.8</td>
</tr>
<tr>
<td>1:15</td>
<td>0.5939</td>
<td>0.7358</td>
<td>0.4978</td>
<td>0.8</td>
</tr>
<tr>
<td>1:20</td>
<td>0.0541</td>
<td>0.8207</td>
<td>0.0475</td>
<td>0.9</td>
</tr>
</tbody>
</table>
V. CONCLUSIONS AND FUTURE WORK

We introduce a local-to-global semantic clone detection approach that combines deep learning with classical DP idea. Through experiments, we verify the feasibility and effectiveness of the approach. The approach shows that deep neural networks can be used to extract key features in the code, whether at the textual, syntactic or semantic level. This not only shows that deep learning can be applied to clone detection, but also shows that deep learning techniques can be used as an effective tool for understanding code, and code understanding plays a key role in various types of software engineering problems.

Here we present some of the shortcomings of our approach, which are also recommendations for future work. First of all, our approach is not limited to a certain programming language. However, while Soot provides us with an excellent and convenient way to generate CFGs that unify locally identical semantics, it also limits our current experiments to the Java language. If some cross-language CFG generation approaches can be added, this approach can be easily extended to a cross-language clone detection approach. Second, the time complexity of the DTW algorithm is \( O(n^2) \), so although our approach is acceptable for detecting most large-scale software systems, there is still much room for improvement in terms of time efficiency. On the one hand, some improved DTW algorithms can be used. On the other hand, some filtering rules can be added before the actual detection to remove code pairs that are obviously not a clone pair. Finally, although our approach can achieve good results in detecting local and global semantic clones, we have not combined them. How to combine local and global semantic clone detection to complete a one-stop semantic clone detection is also an interesting question.

REFERENCES


