CC2Vec: Combining Typed Tokens with Contrastive Learning for Effective Code Clone Detection

SHIHAN DOU, Fudan University, China
YUEMING WU*, Nanyang Technological University, Singapore
HAOXIANG JIA, Huazhong University of Science and Technology, China
YUHAO ZHOU, Fudan University, China
YAN LIU, Fudan University, China
YANG LIU, Nanyang Technological University, Singapore

With the development of the open source community, the code is often copied, spread, and evolved in multiple software systems, which brings uncertainty and risk to the software system (e.g., bug propagation and copyright infringement). Therefore, it is important to conduct code clone detection to discover similar code pairs. Many approaches have been proposed to detect code clones where token-based tools can scale to big code. However, due to the lack of program details, they cannot handle more complicated code clones, *i.e.,* semantic code clones. In this paper, we introduce *CC2Vec*, a novel code encoding method designed to swiftly identify simple code clones while also enhancing the capability for semantic code clone detection. To retain the program details between tokens, *CC2Vec* divides them into different categories (*i.e.,* typed tokens) according to the syntactic types and then applies two self-attention mechanism layers to encode them. To resist changes in the code structure of semantic code clones, *CC2Vec* performs contrastive learning to reduce the differences introduced by different code implementations. We evaluate *CC2Vec* on two widely used datasets (*i.e.,* BigCloneBench and Google Code Jam) and the results report that our method can effectively detect simple code clones. In addition, *CC2Vec* not only attains comparable performance to widely used semantic code clone detection systems such as *ASTNN*, *SCDetector*, and *FCCA* by simply fine-tuning, but also significantly surpasses these methods in both detection efficiency.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools.*

Additional Key Words and Phrases: Code clone detection, contrastive learning, self-attention mechanism

ACM Reference Format:
Shihan Dou, Yueming Wu, Haoxiang Jia, Yuhao Zhou, Yan Liu, and Yang Liu. 2024. CC2Vec: Combining Typed Tokens with Contrastive Learning for Effective Code Clone Detection. *Proc. ACM Softw. Eng.* 1, FSE, Article 70 (July 2024), 21 pages. https://doi.org/10.1145/3660777

1 Introduction

Code clone, also known as duplicate code or similar code, refers to two or more identical or similar source code fragments that exist in the code base. According to the syntactic or semantic level differences, code clones can be classified into four types [15, 42]. Type-1 to Type-3 code clones

*Yueming Wu is the corresponding author.

Authors' Contact Information: Shihan Dou, Fudan University, Shanghai, China, shdou21@m.fudan.edu.cn; Yueming Wu, Nanyang Technological University, Singapore, Singapore, wuyueming21@gmail.com; Haoxiang Jia, Huazhong University of Science and Technology, Wuhan, China, haoxiangjia@hust.edu.cn; Yuhao Zhou, Fudan University, Shanghai, China, zhouyh21@m.fudan.edu.cn; Yan Liu, Fudan University, Shanghai, China, yliu22@m.fudan.edu.cn; Yang Liu, Nanyang Technological University, Singapore, Singapore, yangliu@ntu.edu.sg.

ACM 2994-970X/2024/7-ART70
https://doi.org/10.1145/3660777

belong to syntactic code clones, while Type-4 code clones are semantic code clones. Most of the existing code clone detection methods aim to detect syntactic code clones because these clones are easier to identify compared to semantic code clones. For example, to detect the first two types of code clones, *CCFinder* [26] first performs lexical analysis to extract the token sequence of a code snippet. After obtaining all tokens, it conducts several code transformations to convert the tokens into corresponding forms and then uses the transformed tokens to finish code clone detection. Since *CCFinder* [26] only applies certain simple code transformations, it cannot detect Type-3 code clones. To address the issue, other token-based methods are designed by analyzing the token features between the two codes within a pair from different perspectives. For example, *SourcererCC* [44] computes the overlap ratio of tokens to detect near-miss Type-3 code clones. However, due to the lack of preservation of program semantics, these token-based methods do badly in detecting semantic code clones since the code structure may change a lot after using another way of implementation. For example, when implementing the same loop-related functionality, some developers may use *for* statements while others may prefer to use *while* statements.

To tackle the issue, researchers perform complex program analysis to transform the code fragments into different intermediate representations such as *abstract syntax tree* (AST) [61] and *control flow graph* (CFG) [59]. These intermediate representations can maintain the program semantics from different perspectives. Empirical experiments have demonstrated the capability of tree-based and graph-based methods [28, 30, 51] on detecting semantic code clones. However, both tree processing and graph analysis are time-consuming, making it difficult for them to find semantic code clones efficiently. A recent report [4] has shown that open source has become an unstoppable trend, leading to an increasing scale of code reuse. These cloned codes can bring uncertainty and risks to the software system, such as bug propagation, and copyright infringement [5]. Therefore, there is an urgent need to develop a tool that can be used for large-scale semantic clone detection.

In this paper, we propose *CC2Vec*, a novel code encoding approach to efficiently detect syntactic code clones, while enhancing the ability of semantic code clone detection by further combining with a few neural networks. Specifically, we mainly address two challenges.

- *Challenge 1: How to retain the program semantics by purely analyzing the source code tokens?*
- *Challenge 2: How to differentiate codes based on their functionalities instead of their structures?*

To address the first challenge, we parse the source code into corresponding tokens and divide them into 15 categories based on syntactic types by lexical analysis. We call tokens in different categories as *typed tokens* and observe that these typed tokens have different weights in detecting code clones (Details are in Section 2). Based on the observation, we first apply one self-attention mechanism layer to encode the tokens in the same category into a vector representation. By this, the source code is converted into 15 vectors, each corresponding to a token category. After obtaining 15 vectors, we then apply another self-attention mechanism layer to encode them into one vector, which is the output of this phase. Using self-attention mechanism layers can not only interpret the detection results but also preserve the potential relationships between tokens. These relationships may represent some program details of a method.

To solve the second challenge, we conduct contrastive learning to train the program encoder. The use of contrastive learning is to maximize the similarity between representations of a code snippet and its clone codes (*i.e.,* positive samples), while minimizing the representations' similarity between this code snippet and its non-clone codes (*i.e.,* negative samples). As for semantic code clones, although their code structures may change a lot, the implemented functionality remains unchanged. In other words, they can be treated as clone samples of each other. Therefore, we can use contrastive learning to decrease the distance between semantic code pairs while enlarging the difference between dissimilar code pairs. After contrastive learning, the learned encoder *CC2Vec* is robust to code changes introduced by semantic code cloning, making it possible to distinguish semantic code clones although they are syntactically different.

We design a token-based code clone detector by simply calculating the cosine similarity of the two codes' vector encoded by *CC2Vec*, to detect syntactic code clones efficiently. In addition, we can effectively detect more complicated code clones (*i.e.,* semantic code clones) by combining *CC2Vec* with a few neural networks. To examine the ability of *CC2Vec*, we conduct evaluations on two widely used datasets, namely BigCloneBench [3, 46] and Google Code Jam [2]. Compared to five pretrain-based methods, *CC2Vec* can effectively capture the tokens relationship information by pre-training from the training code corpus to achieve an improvement in detecting code clones. Compared to nine traditional code clone detectors, the recall of *CC2Vec* is 64% while the comparative tools can only maintain very low recall, respectively. Compared to six deep-learning-based code clone detectors, *CC2Vec* can achieve the best F1 score when using only a simple three-layer neural network as the classifier. As for scalability, *CC2Vec* outperforms most pretrain-based and deep learning-based methods in terms of runtime efficiency. *CC2Vec* is about 100 times faster than *ASTNN* in predicting code clone pairs [1].

In summary, this paper makes the following contributions:

- We introduce a novel encoding approach *CC2Vec* to enhance the ability on code clone detection by using contrastive learning. *CC2Vec* applies two self-attention mechanisms layers to encode source code tokens and contrastive learning to learn a robust encoder.
- We improve the detection accuracy of syntactic code clones by simply computing the cosine similarity of two codes' vectors encoding by *CC2Vec*. Additionally, we can also effectively detect the semantic code clones by combining *CC2Vec* with a few neural networks.
- We check the ability of *CC2Vec* by conducting experiments on two widely used datasets (*i.e.,* BigCloneBench and Google Code Jam). Experimental results show that our proposed detectors based on *CC2Vec* can achieve comparable performance to widely used code clone detectors with efficient training and detecting phase.

Paper organization. The remainder of the paper is organized as follows. Section 2 presents our background. Section 3 introduces our system. Section 4 reports the experimental results. Section 5 shows our discussions. Section 6 describes the related work. Section 7 concludes the present paper.

2 Background

In this section, we first discuss four types of code clones and then clarify the key insight of *CC2Vec*.

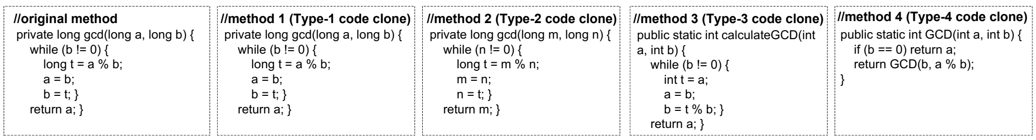
2.1 Clone Types

As aforementioned, code clones can be divided into four types [15, 42]. The first three types belong to syntactic code clone while the last type is semantic code clone.

- **Type-1**: The same code snippets, except for differences in white space and comments.
- **Type-2**: The same code snippets, except for differences in identifier names, literal values, white space, and comments.
- **Type-3**: Syntactically similar code snippets that differ at the statement level. The snippets have statements added, modified and/or removed.
- **Type-4**: Syntactically dissimilar code snippets that implement the same functionality.

To illustrate the differences between four types of code clones, we present one method and its four types of code clones in Figure 1. They implement the same functionality which is to calculate the greatest common divisor of two numbers. Method 1 is the Type-1 code clone of the original method, with no changes between their source code. Method 2 is the Type-2 code clone of the

[1] Our tool is available on our website: *https://github.com/CC2Vector/CC2Vec*.



```
//original method
private long gcd(long a, long b) {
    while (b != 0) {
        long t = a % b;
        a = b;
        b = t; }
    return a; }
```

```
//method 1 (Type-1 code clone)
private long gcd(long a, long b) {
    while (b != 0) {
        long t = a % b;
        a = b;
        b = t; }
    return a; }
```

```
//method 2 (Type-2 code clone)
private long gcd(long m, long n) {
    while (n != 0) {
        long t = m % n;
        m = n;
        n = t; }
    return m; }
```

```
//method 3 (Type-3 code clone)
public static int calculateGCD(int
a, int b) {
    while (b != 0) {
        int t = a;
        a = b;
        b = t % b; }
    return a; }
```

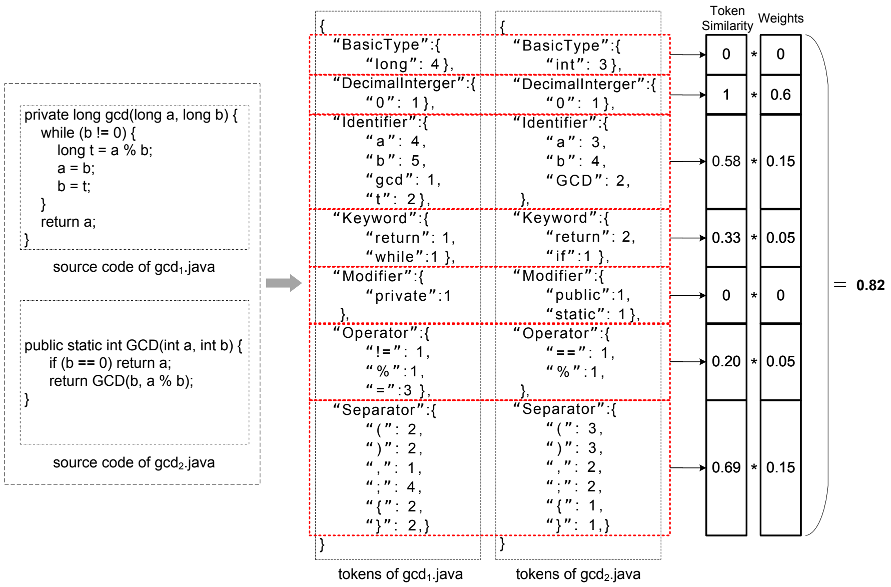
```
//method 4 (Type-4 code clone)
public static int GCD(int a, int b) {
    if (b == 0) return a;
    return GCD(b, a % b);
}
```

Fig. 1. Examples of four code clone types

original method, and the difference between them is only in the identifiers name (*i.e., m* and *n* instead of *a* and *b*). The few code changes make them easy to detect. Method 3 is the Type-3 code clone of the original method, it differs at the statement level. Their method names and parameter types are different and the order of the statements also changes a little. This type of code clone is more difficult to detect than the previous two types. Method 4 is the Type-4 code clone of the original method, it implements the same functionality in a different way. This type of clone is also called semantic clone and is the most difficult to discover since the code structure may change a lot.

2.2 Motivation

We propose an example to illustrate the key insight of our method. This example is a clone pair in BigCloneBench [3], it consists of two methods and their functionalities are both to compute the greatest common divisor of two integers. As shown in Figure 2, although they are syntactically dissimilar, the functionalities are the same. Therefore, they can be treated as a semantic clone pair.



```
private long gcd(long a, long b) {
    while (b != 0) {
        long t = a % b;
        a = b;
        b = t;
    }
    return a;
}
```

source code of gcd₁.java

```
public static int GCD(int a, int b) {
    if (b == 0) return a;
    return GCD(b, a % b);
}
```

source code of gcd₂.java

```
{
  "BasicType":{
      "long": 4},
  "DecimalInterger":{
      "0": 1},
  "Identifier":{
      "a": 4,
      "b": 5,
      "gcd": 1,
      "t": 2},
  "Keyword":{
      "return": 1,
      "while":1 },
  "Modifier":{
      "private":1
  },
  "Operator":{
      "!=": 1,
      "%":1,
      "=":3 },
  "Separator":{
      "(": 2,
      ")": 2,
      ",": 1,
      ";": 4,
      "{": 2,
      "}": 2,}
}
```

tokens of gcd₁.java

```
{
  "BasicType":{
      "int": 3},
  "DecimalInterger":{
      "0": 1},
  "Identifier":{
      "a": 3,
      "b": 4,
      "GCD": 2,
  },
  "Keyword":{
      "return": 2,
      "if":1 },
  "Modifier":{
      "public":1,
      "static": 1},
  "Operator":{
      "==": 1,
      "%":1,
  },
  "Separator":{
      "(": 3,
      ")": 3,
      ",": 2,
      ";": 2,
      "{": 1,
      "}": 1,}
}
```

tokens of gcd₂.java

Token Similarity		Weights
0	*	0
1	*	0.6
0.58	*	0.15
0.33	*	0.05
0	*	0
0.20	*	0.05
0.69	*	0.15

= **0.82**

Fig. 2. Source code and corresponding tokens of *gcd*$_1$*.java* and *gcd*$_2$*.java* after lexical analysis

As aforementioned, *SourcererCC* [44] is one of the most scalable token-based code clone detector, which can scale to analyze more than 428 million files on Github [35]. It uses a simple overlap similarity calculation to measure the distance between two methods. For example, given two methods M_1 and M_2, the overlapping similarity $S(M_1, M_2)$ is computed as the quotient of the number of same tokens shared by M_1 and M_2 and the maximum number of tokens in M_1 and M_2, which can be written as $S(M_1, M_2) = \frac{|M_1 \bigcap M_2|}{max(|M_1|, |M_2|)}$.

To compute the overlapping similarity of methods in List 1 and List 2, we first perform lexical analysis to collect the corresponding source code tokens. Then the number of the same tokens shared by these two methods is obtained by statistical analysis. After completing the analysis, we find that there are 19 tokens shared by gcd_1*.java* and gcd_2*.java*. Therefore, the similarity computed by *SourcererCC* is 0.5 (*i.e.,* 19/38=0.5) since the maximum number of tokens of gcd_1*.java* and gcd_2*.java* is 38. Because 0.5 is smaller than 0.7 which is the default similarity threshold in *SourcererCC*, gcd_1*.java* and gcd_2*.java* will not be reported as a clone pair. In other words, it will be a false negative when we use *SourcererCC* to detect gcd_1*.java* and gcd_2*.java*.

To detect the semantic code pair, we conduct further analysis on their source code tokens. We observe that these tokens can be divided into different categories. Figure 2 presents the tokens in different categories of gcd_1*.java* and gcd_2*.java* after applying lexical analysis. Through the results, we can see that the tokens in some categories are similar, while the tokens in certain categories are different. For example, only four "*long*" tokens belong to *BasicType* category in gcd_1*.java*, while the tokens belonging to *BasicType* category in gcd_2*.java* are three "*int*" tokens. As for tokens in *DecimalInterger* category, gcd_1*.java* and gcd_2*.java* are the same, that is, only one "*0*" token. To measure the similarity of tokens in different categories, we apply Eq. **??** to the tokens within the single category to compute the category-level overlapping similarity of these two methods. To measure the similarity of tokens in different categories, we compute the above category-level overlapping similarity of these two methods for each token category. For instance, the overlapping similarity for the *Separator* category can be calculated as 0.69 (*i.e.,* (2+2+1+2+1+1)/max((2+2+1+4+2+2), (3+3+2+2+1+1)) = 9/13 = 0.69). After computing all similarities, we find that they are different in different categories. Some categories have high similarities, while others have low similarities. In other words, if we assign a higher weight to a category with high similarity and a smaller weight to a category with low similarity, then gcd_1*.java* and gcd_2*.java* are likely to be detected as a clone pair. For example, if the weights of seven categories in Figure 2 are 0, 0.6, 0.15, 0.05, 0, 0.05, and 0.15, respectively. The similarity will be calculated as 0.82 (*i.e.,* $0*0 + 1*0.6 + 0.58*0.15 + 0.33*0.05 + 0*0 + 0.20*0.05 + 0.69*0.15 = 0.82$. In each multiplication, the first value represents the category overlap similarity, and the second value denotes its assigned weight) which is higher than the default similarity threshold (*i.e.,* 0.7) in *SourcererCC*.

Therefore, inspired by the observation, we introduce the attention mechanism to design a novel technique to train an encoder that can assign suitable weights for different categories, to detect all kinds of code clones. The attention mechanism can automatically focus on important tokens, which has the potential to filter the useless tokens and categories in detecting code clones, while concentrating the attention on these useful tokens and categories.

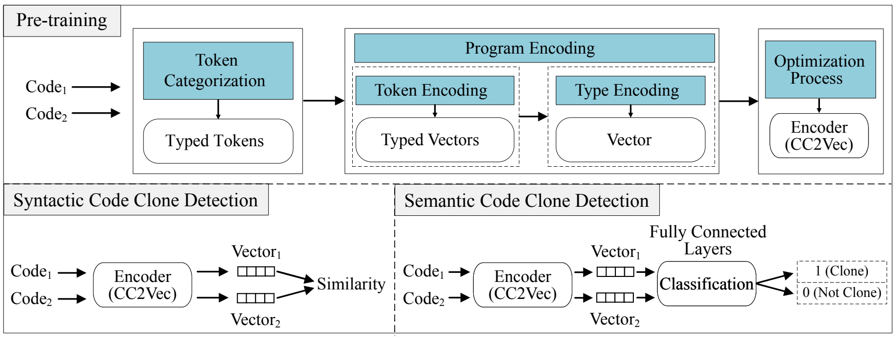


Fig. 3. System of *CC2Vec*

3 System

In this section, we introduce our code encoding approach, namely *CC2Vec*, and how to utilize it to detect syntactic and semantic code clones.

3.1 *CC2Vec* Overview

As shown in Figure 3, *CC2Vec* is made up of three main components: *Token Categorization*, *Program Encoding*, and *Optimization Process*.

- ***Token Categorization***: Given the source code of a method, we first apply lexical analysis to parse them into tokens with corresponding categories, namely *typed tokens*. The input of this component is a method, while the output is certain typed tokens.
- ***Program Encoding***: This component consists of two attention mechanism layers. The first layer encodes tokens in each category into a vector, and the second layer encodes all categories into the final vector. The input of this component is certain typed tokens, while the output is a vector representation.
- ***Optimization Process***: In this phase, we employ contrastive learning as the optimization approach to train the program encoder which is robust to code changes introduced by different implementations.

Table 1. Selected 15 token types

"*Annotation*", "*BasicType*", "*BinaryInteger*", "*Boolean*", "*DecimalFloatingPoint*", "*Modifier*", "*Operator*", "*DecimalInteger*", "*HexFloatingPoint*", "*HexInteger*", "*Identifier*", "*Keyword*", "*OctalInteger*", "*Separator*", "*Null*"

3.2 Token Categorization

This component aims to parse the source code of a method into tokens with corresponding categories (*i.e., typed tokens*). To complete the purpose, we conduct a lexical analysis to analyze the source code to divide it into different categories based on syntactical types. Since the experimental dataset is BigCloneBench [3] and the programming language is *Java*, we leverage a python library *javalang* [6] to accomplish the lexical analysis of *CC2Vec*. In practice, token categorization aims to divide a method into *typed tokens*, which can be achieved by lexical analysis. Therefore, *CC2Vec* is not limited to programming languages (*e.g., C* and *Java*) because different languages have related lexical analysis tools (*e.g., pycparser* [8] for *C* language and *javalang* [6] for *Java* language).

In practice, listing all token types parsed by *javalang* can be challenging due to the potential existence of some types that are used infrequently. To identify the most relevant token types, we conducted a lexical analysis on the top 10,000 Java projects on GitHub, ranked by their criticality score. This score can be used to describe the impact and importance of an open source project [10]. After our statistical analysis, we found that 14 types collectively constitute over 99.5% of all tokens. Therefore, we choose these 14 types as the final token types and add a *Null* type to represent other types as shown in Table 1.

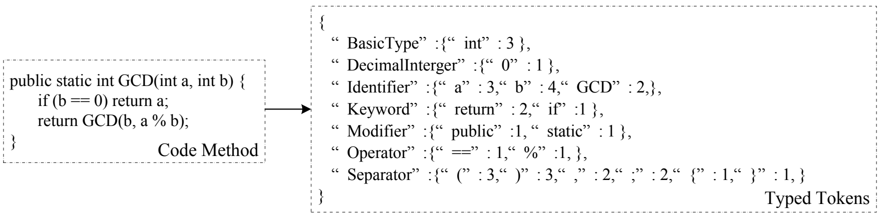

```
public static int GCD(int a, int b) {
    if (b == 0) return a;
    return GCD(b, a % b);
}
```
```
{
  " BasicType" :{" int" : 3 },
  " DecimalInterger" :{" 0" : 1 },
  " Identifier" :{" a" : 3," b" : 4," GCD" : 2,},
  " Keyword" :{" return" : 2," if" :1 },
  " Modifier" :{" public" :1, " static" : 1 },
  " Operator" :{" ==" : 1," %" :1, },
  " Separator" :{" (" : 3," )" : 3," ," : 2," ;" : 2," {" : 1," }" : 1, }
}
```

Fig. 4. Token categorization of *CC2Vec*

To better describe the different steps of our system, we present one simple example in Figure 4. It shows that tokens of the method in List 2 can be divided into seven categories after applying lexical analysis and each category contains the corresponding tokens and the number of tokens. For example, in "*BasicType*" category, it only contains one token "*int*", and the number of tokens "*int*" is three. The output of this component is tokens with corresponding categories, namely *typed tokens*. Since different tokens may determine whether two pieces of code are clones, as discussed in Section 2, if we can identify the optimal tokens and increase their weights automatically, we will be able to detect code clones more effectively.

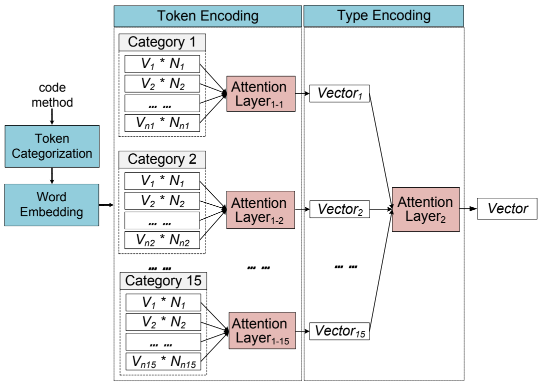


Fig. 5. Program encoding of *CC2Vec*

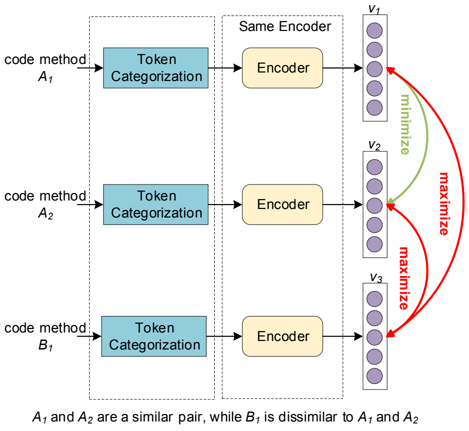


Fig. 6. Contrastive learning of *CC2Vec*

3.3 Program Encoding

This component aims to encode *typed tokens* into a vector representation. As shown in Figure 5, it consists of two steps: *Token Encoding* and *Type Encoding*.

3.3.1 Token Encoding. This step aims to encode tokens within a single category into vector representations. To achieve this, we initially employ a word embedding technique to map tokens to their corresponding vector representations. Subsequently, these vectors are input into an attention mechanism layer for analysis. Specifically, to encode tokens with similar meanings into high-dimensional feature spaces with comparable distances, we utilize the source code from BigCloneBench [3] as the training set to train a *Word2Vec* model [37]. *Word2Vec* stands as one of the most widely used techniques for learning word embeddings through shallow neural networks. After obtaining corresponding vector representations of all tokens in one category, we multiply them by the number of corresponding tokens one by one. These multiplied vectors will be fed into one attention mechanism layer to be encoded. Note that each category is assigned an attention mechanism layer, and the layers are independent of each other. In other words, we design 15 independent attention mechanism layers to encode tokens in 15 categories, respectively. Some semantic relationships exist between statements in a method (*e.g.,* , data flow), resulting in potential relationships between tokens. To consider these potential relationships while program encoding, we use the self-attention mechanism as our encoding layer since it can not only describe the importance or weights but also consider the relationships between tokens. Our self-attention mechanism layer consists of one multi-head attention layer and one feed-forward layer. More details of them are shown in *Transformer* [49]. After token encoding, each category can output a vector.

3.3.2 Type Encoding. This step aims to encode vectors obtained from *Token Encoding* step into a vector representation. Due to some potential relationships between tokens, there are some relationships between categories. Therefore, we also apply one self-attention mechanism layer to finish the final encoding. It is worth noting that some methods might not encompass all 15 categories. For example, after lexical analysis, the method in Figure 4 can only be divided into seven categories. To ensure appropriate weight allocation across all categories, we assign a value of 0 to vectors corresponding to absent categories. Setting each position of absent types of token to zero, instead of category 0 is intended to ensure that the results of subsequent matrix operations in attention are 0, preventing this token from interfering with others. By this, the inputs of *Type Encoding* step are 15 vectors corresponding to 15 categories. The self-attention mechanism layer used in this step is the same as that used in *Token Encoding* step. After *Type Encoding*, we can obtain the final vector representation of a given method.

3.4 Optimization Process

Given a dollar bill, it is difficult for us to draw it exactly the same, although we have seen it many times. However, we can easily use our drawing to convey the essence of the dollar bill to others [12]. Drawing inspiration from such commonplace observations, researchers have designed a learning algorithm. Instead of focusing on every sample detail, it extracts enough abstract features to distinguish it from other samples. This learning algorithm is also known as contrastive learning, its goal is to make the distance between different types of inputs get larger and larger, and the same types of inputs get closer and closer. Since contrastive learning only needs to learn to distinguish the sample in the feature space of the abstract semantic level, the model and its optimization become simpler and the generalization ability is stronger. Formally, contrastive learning aims to allow the encoder to generate more unique data features without losing its essence. For any data sample x and its feature $f(x)$, it intends to complete a task as equation (1).

$$\mathrm{SIM}(f(x), f(x^{+})) >> \mathrm{SIM}(f(x), f(x^{-})) \tag{1}$$

Here x^{+} is a data sample similar to x (referred to as the positive sample) and x^{-} is a data sample dissimilar to x (referred to as the negative sample). $f(x^{+})$, $f(x^{-})$ is the feature of x^{+}, x^{-} separately, and $\mathrm{SIM}(\cdot)$ is a metric to measure the similarity between two features.

We utilize contrastive learning as the optimization method to train the program encoder. Specifically, the model training and optimization process is shown in Figure 6, where code method A_1 and code method A_2 are a clone pair, and code method B_1 is dissimilar to code methods A_1 and A_2. Therefore, the optimization goal of the model is to reduce the distance between the encoded vectors of the code method and its similar pairs (*i.e.,* positive samples), such as A_1 and A_2, and increase the distance between the encoded vectors of the code method and its dissimilar pairs (*i.e.,* negative samples), such as A_1 and B_1. As for semantic code clones, although their code structures may change a lot, the implemented functionality remains unchanged. Therefore, they can be treated as positive samples of each other.

3.5 Code Clone Detection

After completing the training phase by using contrastive learning, the learned encoder can be robust to code changes introduced by different implementations, making it possible to identify the subtle difference between the two code snippets. The code encoder can be directly used for discovering the simple syntactic code clones, while can detect semantic code clones by continuing fine-tuning.

Firstly, *CC2Vec* introduces two self-attention mechanisms layers to capture potential relationships between tokens, which means that it can encode more important information. Therefore, we can directly design a simple code clone detector to identify code clones efficiently, and easily to scale to big code. Specifically, for the Type-1 to Type-3 code clones (*i.e.,* syntactic code clones), we obtain the two code representations by using *CC2Vec* and calculate the cosine similarity between these two vectors. If their cosine similarity is higher than 70% [44], they will be reported as a clone pair. Otherwise, they will be treated as a non-clone pair.

Secondly, for the Type-4 code clones (*i.e.,* semantic code clones), we combine *CC2Vec* with a few neural networks (*i.e.,* fully connected layer) to enhance the ability to detect more complicated samples. As *CC2Vec* has learned potential information from code data, fine-tuning it with one to three fully connected layers can significantly improve the detection ability on semantic code clones. Specifically, we obtain the vectors corresponding to each of the two code snippets within a pair and then concatenate them to form a single vector representation. This vector is then input to multiple fully connected layers to detect whether the code pair is a clone pair or not. For instance, *CC2Vec-3L* denotes that we connect our pre-trained encoder *CC2Vec* with three fully connected layers. Notably, we insert a ReLu activation layer between each fully connected layer to ensure the non-linear capability of the neural networks.

4 Experiments

In this section, we undertake comprehensive experiments to assess *CC2Vec*'s ability to detect code clones from three key perspectives: effectiveness, interpretability, and scalability. For effectiveness, we not only evaluate *CC2Vec*'s performance in detecting both syntactic and semantic code clones but also demonstrate the contributions of its different components (*i.e.,* self-attention and contrastive learning) to code clone detection. For interpretability, we provide examples to illustrate how *CC2Vec* interprets the detection results, shedding light on its decision-making process. For scalability, we present a detailed experiment to show the runtime overhead of *CC2Vec*, providing insights into its computational efficiency. Specifically, we aim to answer the following research questions:

- *RQ1: Can CC2Vec detect code clones effectively?*
- *RQ2: How do self-attention mechanism layers and contrastive learning contribute to CC2Vec?*
- *RQ3: Can CC2Vec interpret the detection results?*
- *RQ4: Can CC2Vec scale to big code?*

4.1 Experimental Settings

4.1.1 Dataset. To check the ability of *CC2Vec* to detect code clones, we choose BigCloneBench [3] as our first dataset since it is the largest dataset and widely used by many researchers [44, 52, 61]. Experts assign all pairs in BigCloneBench a clone type according to a similarity score calculated by line-level and token-level after code normalization. As for Type-3 and Type-4 code clones, they are divided into four parts due to the ambiguous boundary between them, that is, 1) *Very Strongly Type-3* (VST3) with a similarity between [0.9, 1.0), 2) *Strongly Type-3* (ST3) with a similarity between [0.7, 0.9), 3) *Moderately Type-3* (MT3) with a similarity between [0.5, 0.7), and 4) *Weakly Type-3/Type-4* (WT3/T4) with a similarity between [0.0, 0.5). BigCloneBench contains about 270,000 non-clone pairs and over eight million tagged clone pairs. Finally, we obtain 48,116 T1 clones, 4,234 T2 clones, 4,577 VST3 clones, 16,818 ST3 clones, 86,341 MT3 clones and 7,943,729 WT3/T4 clones. To balance our dataset, we select almost 270, 000 clone pairs from them. Specifically, we select all of the syntactic pairs to balance the number of each code clone pair type, and randomly select 109,914 clone pairs from a total of about eight million semantic clones. As a result, there are 48,116 T1 clones, 4,234 T2 clones, 4,577 VST3 clones, 16,818 ST3 clones, 86,341 MT3 clones, and 109,914 WT3/T4 clones in our dataset.

Similar to previous work [59], for the more challenging semantic clones, we also evaluate our proposed method on another widely used dataset namely Google Code Jam [62], which is primarily composed of semantic clones, to better demonstrate the effectiveness of our method. Google Code Jam is derived from an online programming competition held by Google and contains 1,669 projects from 12 different competition problems which are written by different programmers. So projects of the same competing problem are almost syntactically different but semantically similar, and we treat them as clone pairs. Projects that solve different problems are not similar, and we regard them as non-clone pairs. Finally, we obtain 275,570 semantic clone pairs and 1,116,376 non-clone pairs. We randomly select 270,000 pairs from all non-clone pairs to balance our dataset.

4.1.2 Implementations. For detecting the syntactic code clones, we only need to train a code encoder *CC2Vec*. We use *Word2Vec* [37] to embed the token to the vector. The size of the embedding layer and the hidden size of the two attention layers are 100. We average the hidden cell of the attention layer to obtain the output vector, so the vector size is also 100. During the pre-trained phase, we freeze the embedding layer and only the attention layer is trainable. For contrastive learning, we apply the SupCon loss function [27] as the training objective. The temperature in this loss function is set to 0.07. During the training phase, we use the RMSProp optimization algorithm [1], with a momentum of 0.9 and a weight decay of 0.0001, to train the encoder. The learning rate and the training epochs are set to 0.0001 and 10, respectively. The training batch size is 64. After training, we can obtain a program encoder to extract high-level potential features from code snippets. The threshold is set to 0.7, based on the previous research [44], indicating that if the cosine similarity between the two code snippets within a pair exceeds 70%, they are classified as a clone pair. Otherwise, they are considered a non-clone pair. For fully connected layers used to enhance the ability to detect semantic code clones, we need to fine-tune the code encoder *CC2Vec* and the newly introduced neural networks. The learning rate and optimizer are the same as above. Notably, the training dataset utilized is the same dataset used in training the encoder *CC2Vec*, to avoid any leakage of validation and test data into the training process.

4.1.3 Comparative Tools. We also compare *CC2Vec* with some state-of-the-art code clone detection systems. The choice of our baseline follows these principles: The method is popular and extensively used, with a high citation in the code clone detection and widely compared in other works. Specifically, we choose five pretrain-based methods (*i.e., Word2Vec* [37], *Doc2Vec* [32], *Code2Vec* [11], *CodeBERT* [20], and *CodeT5* [53]), nine traditional code clone detectors (*i.e., SourcererCC* [44], *CCFinder* [26], *Nicad* [41], *Decakrd* [25], *CCAligner* [52], *Oreo* [43], *LVMapper* [57], *NIL* [40], and *Tamer* [22]), and six deep-learning-based code clone detectors (*i.e., RtvNN* [56], *CDLH* [54], *TBCNN* [38], *ASTNN* [61], *SCDetector* [59], and *FCCA* [24]). Note that we use the parameters recommended in each paper for these works, and the experimental results are also close to the best result reported in their paper to ensure credibility.

4.1.4 Experimental Environment and Metrics. For token categorization, we make use of a Python library (*i.e., javalang* [6]) to complete our lexical analysis. After parsing the source code into typed tokens, we leverage *Pytorch* to implement the self-attention mechanism layers and contrastive learning to train the encoder. Given two vectors of two methods, we use another Python library (*i.e., Sklearn* [7]) to compute the cosine similarity. To measure the effectiveness of *CC2Vec*, we adopt the following widely used metrics. Precision is defined as $P = TP/(TP + FP)$. Recall is defined as $R = TP/(TP+FN)$. F1 is defined as $F1 = 2*P*R/(P+R)$. Among them, *true positive* (TP) represents the number of samples correctly classified as clone pairs, *false positive* (FP) represents the number of samples incorrectly classified as clone pairs, and *false negative* (FN) represents the number of samples incorrectly classified as non-clone pairs. We employ ten-fold cross-validation to record the F1 score, precision, and recall for each validation run on all experiments. We calculate the average of these metrics across the ten validations and consider this average as the final performance.

4.2 RQ1: Detection Effectiveness

In this subsection, we evaluate our proposed system by comparing it to three types of code clone detectors, *i.e.,* pretrain-based methods, traditional scalable methods, and deep learning-based methods. Firstly, we assess *CC2Vec*'s ability to detect syntactic code clones compared to other popular pretrain-based methods and traditional scalable tools. Secondly, we fine-tune *CC2Vec* with neural networks and evaluate its semantic code clones detection effectiveness compared to widely used deep learning-based detectors.

4.2.1 Performance of CC2Vec Compared to Pretrain-based Methods. We first analyze our method's effectiveness in detecting syntactic code clones compared to pretrain-based methods. Similar to most of the code clone detectors [44], when we set the detection threshold of *CC2Vec* to 0.7, the precision under this threshold is 98%. At the same time, we find that the detection thresholds of different pretrain-based methods are not the same, and to ensure a reasonable comparison, we adapt the thresholds of the other detection methods to make all of them have an accuracy rate of 98%. The threshold of *Word2Vec*, *Doc2Vec*, *Code2Vec*, *CodeBERT*, and *CodeT5* is 0.85, 0.75, 0.91, 0.95 and 0.87, respectively. At the same time, we analyze the recall on cloned code pairs to evaluate the ability of these pretrain-based methods.

Table 2. Results of pretrain-based methods and our method *CC2Vec* on BigCloneBench dataset

Methods	**Recall**						**Precision**
	T1	**T2**	**VST3**	**ST3**	**MT3**	**T4**	
Word2Vec	1	1	**0.99**	0.85	0.53	0.1	0.98
Doc2Vec	1	0.95	0.86	0.57	0.21	0.02	0.98
Code2Vec	1	1	0.95	0.82	0.54	0.17	0.98
CodeBERT	1	1	0.95	0.89	0.71	0.23	0.98
CodeT5	1	1	0.98	0.91	0.77	0.49	0.98
CC2Vec	**1**	**1**	0.97	**0.93**	**0.81**	**0.64**	**0.98**

As shown in Table 2, all methods achieved good performance on easier-to-find clones like Type-1, Type-2, and *Very Strongly Type-3* (VST3), being able to detect and recall the vast majority of them. However, we also see that the recall scores of other pretrain-based methods on *Moderately Type-3* (MT3) and Type-4 perform poorly and are far lower than their precision scores. Although these code clone detectors model the code language through code corpus, they do not capture the semantic information in the pre-training corpus, but only learn the surface token-based information of the code and obtain an efficient method representation. The recall scores of *CodeBERT* and *CodeT5* on Type-4 clones are 23% and 49%, respectively, which indicates that they are capable of detecting Type-4 code clones. This is reasonable since *CodeBERT* and *CodeT5* are pre-trained on a massive code corpus [17, 20, 53] and have a much higher number of parameters than the other four pretraining-based methods. On the other hand, despite *CodeBERT* and *CodeT5* having a large number of parameters and being pre-trained on extensive code corpora, they do not specifically incorporate the code clone detection task into pre-training tasks, nor do they design optimization goals for code clone detection. As a result, they face difficulty in accurately detecting Type-4 code clones.

For *CC2Vec*, the experimental results show that the recall score on Type-4 clones is 64% with 98% precision, achieving the best performance in pretrain-based methods on BigCloneBench datasets. This is reasonable that *CC2Vec* considers the potential relationships between tokens, thereby extracting more information than other vanilla pre-trained models during the training phase. Meanwhile, contrastive learning provides a better way to learn the information between the code snippets and its clone code snippets (*i.e.,* positive samples) and other non-clone segments (*i.e.,* negative samples) within a batch. Moreover, in contrast to *CodeBERT*, *CC2Vec* do not require a massive pre-training code corpus and a large number of parameters. *CC2Vec* is only pre-trained on the training dataset, such as *Word2Vec* and *Doc2Vec*.

In summary, the results indicate that *CC2Vec* outperforms other pretrained-based methods in detecting syntactic code clones, while providing the capability to detect semantic code clones. However, the ability to detect semantic code clones by only using CC2Vec remains limited.

4.2.2 Performance of CC2Vec Compared to Traditional Tools. To further examine the effectiveness of *CC2Vec* in detecting syntactic code clones, we also evaluate *CC2Vec* compared to nine traditional scalable detection tools (*i.e., SourcererCC* [44], *CCFinder* [26], *NiCad* [41], *Deckard* [25], *CCAligner* [52], *Oreo* [43], *LVMapper* [57], *NIL* [40], and *Tamer* [22]).

The experimental results are described in Table 3. Many detectors achieve perfect or near-perfect recall for T1 and T2 scenarios, while challenges remain for more semantic clone types like Type-3 and Type-4. The results show that *SourcererCC* [44] achieves very low recall scores on MT3 and Type-4 clones. It is reasonable that *SourcererCC* only considers the overlap similarity of tokens between two methods. Therefore, it can not handle semantic clones because it does not consider program semantics. Similar performance effects are found in all other traditional code clone detectors (*i.e.,* the recall of them on Type-4 clones is approximately zero) that are poor at detecting clone pairs at the semantic level (*i.e.,* Type-4 clone pairs). On the other hand, *CC2Vec* can outperform all other traditional methods on MT-3 and Type-4 clones (*i.e.,* the recall is 81% and 64%, respectively) and achieve a high precision (*i.e.,* the precision is 98%).

In summary, these results further indicate that *CC2Vc* can learn more potential relationship information between tokens to detect syntactic code clones, while enhancing a certain ability to detect semantic code clones.

Table 3. Results of the other traditional methods and *CC2Vec* on BigCloneBench dataset

Methods	**Recall**						**Precision**
	T1	**T2**	**VST3**	**ST3**	**MT3**	**T4**	
SourcererCC	1	0.97	0.93	0.6	0.05	0	0.98
CCFinder	1	0.93	0.62	0.15	0.01	0.0	0.72
NiCad	1	0.99	0.98	0.52	0.02	0	**0.99**
Deckard	0.6	0.52	0.62	0.31	0.12	0.01	0.35
CCAligner	1	1	**0.99**	0.65	0.14	0	0.61
Oreo	1	1	1	0.89	0.3	0.01	0.89
LVMapper	1	1	0.98	0.81	0.19	0	0.59
NIL	1	0.97	0.88	0.66	0.19	0	0.94
Tamer	1	1	1	**0.99**	0.53	0.03	0.96
CC2Vec	**1**	**1**	0.97	0.93	**0.81**	**0.64**	0.98

4.2.3 Performance of CC2Vec Compared to Deep Learning-based Code Clone Detectors. To further evaluate our proposed method's efficacy in identifying semantic code clones, we carry out comparative experiments on two extensively used datasets (*i.e.,* BigCloneBench and GoogleCodeJam), against six popular deep learning-based code clone detection techniques (*i.e., RtvNN* [56], *CDLH* [54], *TBCNN* [38], *ASTNN* [61], *SCDetector* [59] and *FCCA* [24]). Notably, we enhance our method's ability to detect semantic clone pairs by integrating the pre-trained encoder *CC2Vec* with several fully connected layers, as described in Section 3.4. The integrated model is subsequently fine-tuned utilizing the identical training dataset employed during the pre-training phase of *CC2Vec*. This approach mitigates any potential leakage of validation and test data into the training process, thereby safeguarding the integrity and reliability of the evaluation.

Table 4 describes the Recall, Precision, and F1 scores of our proposed method and six comparative deep learning-based code clone detectors on two datasets. As in the above experiments, we also use ten-fold cross-validation to record these metrics and report the average scores. *CC2Vec-1L*, *CC2Vec-3L* and *CC2Vec-5L* represent the encoder *CC2Vec* followed by one, three and five fully connected

Table 4. Results of *CC2Vec* and comparative systems for each clone type on BigCloneBench dataset and GoogleCodeJam dataset

Methods	**BigCloneBench**			**GoogleCodeJam**		
	Recall	**Precision**	**F1**	**Recall**	**Precision**	**F1**
RtvNN	0.01	0.95	0.01	0.90	0.20	0.33
CDLH	0.74	0.92	0.82	0.70	0.46	0.55
TBCNN	0.81	0.90	0.85	0.89	0.91	0.90
ASTNN	0.94	0.92	0.93	0.87	0.95	0.91
SCDetector	0.92	0.97	0.94	0.87	0.81	0.82
FCCA	0.92	**0.98**	0.95	0.90	**0.95**	0.92
CC2Vec-1L	0.96	0.92	0.94	0.94	0.91	0.93
CC2Vec-3L	**0.97**	0.94	**0.96**	**0.95**	0.93	**0.94**
CC2Vec-5L	0.97	0.94	0.96	0.95	0.93	0.94

layers, respectively. A ReLU activation layer is inserted in the middle of every two fully connected layers, while for *CC2Vec-1L*, there is no activation layer included. Among the semantic code clone detectors evaluated, *RtvNN*, *CDLH*, *TBCNN* and *ASTNN* are all AST-based clone detection methods. As for *RtvNN*, it applies the RNN model to encode both source code tokens and AST to detect code clones. However, the tree structure may change significantly even when the code changes slightly. The F1 score of *RtvNN* is 1% and 33% for BigCloneBench dataset and GoogleCodeJam dataset, respectively. The other three AST-based methods performed well on both datasets. For example, the F1 of *ASTNN* on the BigCloneBench dataset and GoogleCodeJam dataset achieves 93% and 91%, respectively. Another type of semantic code clone detection tools are graph-based, such as *SCDetector* and *FCCA*. These two graph-based methods also have good detection performance. For the BigCloneBench dataset, the recall of *SCDetector* and *FCCA* is 94% and 82%, respectively. As for the GoogleCodeJam dataset, their recall score is 95% and 92%, respectively.

For *CC2Vec*, when the fully connected layer number is three (*i.e., CC2Vec-3L*), the F1 score is 96% and 94% for BigCloneBench dataset and GoogleCodeJam dataset, respectively, achieving comparable performance to other popular deep learning-based methods. At the same time, we found that increasing the number of fully connected layers to five (*i.e., CC2Vec-5L*) does not result in a significant performance improvement.

In summary, the experimental results demonstrate that through fine-tuning the integrated model, our method can effectively detect all types of code clones including syntactic and semantic code clones. By using token categorization and program encoding, the trainable encoder *CC2Vec* can capture token-level potential relationship information in code segments from code corpus. Meanwhile, contrastive learning enables the encoder can not only learn information from clone pairs, but also learn from non-clone pairs dynamically constructed during the training phase. This learning approach enhances the encoder's ability to identify subtle semantic differences between code snippets, to further achieve an improvement in detecting code clones.

4.3 RQ2: Self-Attention and Contrastive Learning

In this subsection, we conduct three single-factor experiments to validate the use of self-attention mechanism layers and contrastive learning can contribute to the effectiveness of *CC2Vec* on clone detection. Our first experiment adopts two GRU layers to encode the program source code. After training the encoder by using an encoder-decoder model, we can use it to embed other code methods. The output of the learned encoder is the corresponding vector of a method. Given two methods, their cosine similarity will be computed as the standard to detect code clones. If it is higher than 70%, they will be reported as a clone pair. Our second experiment replaces the two GRU layers with two self-attention mechanism layers as the program encoder. However, we do not perform contrastive learning to train the encoder but use an encoder-decoder model to train it.

Similar to the first experiment, we also compute the cosine similarity of the two methods' vectors to check whether they are a clone pair or not. Our third experiment is to implement *CC2Vec*. In other words, we use two self-attention mechanism layers as the program encoder and leverage contrastive learning to train it. The clone detection phase is the same as in the former two experiments. In summary, the difference between the first and second experiments is only in the program encoder, and the difference between the second and third experiments is only in the training phase.

Table 5. Results on BigCloneBench dataset

	Recall	**Precision**	**F1**
Gated Recurrent Unit (GRU)	0.31	0.24	0.27
Self-Attention	0.44	0.75	0.55
Self-Attention + Contrastive Learning (*CC2Vec*)	0.74	0.98	0.84

We describe the results of these three comparative experiments in Table 5. It shows that the use of self-attention mechanism layers can increase the detection accuracy of *CC2Vec*. For example, the F1 score of the first experiment is only 27%, while it increases to 55% when we adopt self-attention mechanism layers to encode source code. It is reasonable because the adoption of self-attention mechanism layers can retain some potential relationships between tokens, and these relationships may contain the program details, resulting in better detection effectiveness in detecting code clones. Moreover, when detecting similar code fragments, self-attention mechanism layers can assign larger weights to similar categories while reducing dissimilar categories' weights. These weights can boost the ability of code clone detection.

Meanwhile, we also find that integrating contrastive learning enhances the robustness of *CC2Vec* to code changes of different implementations. Specifically, when we employ contrastive learning in training the program encoder, the F1 score sees a significant jump to 84%, compared to just 55% without its use. The distance between such semantically similar code pairs is minimized, making it robust to the code changes introduced by different implementations.

In summary, through the results in Table 5, we see that both self-attention mechanism layers and contrastive learning can boost the ability of *CC2Vec* on code clone detection.

4.4 RQ3: Interpretability

Since we leverage self-attention mechanism layers as our program encoder, the assigned weights of different categories can interpret the detection results. As a matter of fact, the dimension of the attention weight matrix is 15×15, corresponding to 15 categories of tokens, and the value of each position (i, j) in the matrix represents the correlation between category i and category j. If the matrix column items are summed, and input into a *Softmax()* layer, the importance (*i.e.,* weight) of each category can be obtained. To validate the interpretability of *CC2Vec*, we use methods in List 1 and List 2 as our examples. Table 6 shows the weights of different token categories in List 1 calculated from the second self-attention layer. Since the tokens of the method in List 1 can only be divided into seven categories, the weights of the other eight categories are all 0.

Through the results in Table 6, we observe that the weight of *BasicType* token category is the largest, which means that tokens in *BasicType* category have the greatest effect in determining whether List 1 and List 2 are code clones or not. To further check whether the weights assigned by *CC2Vec* are suitable or not, we choose tokens in *BasicType* category and *Keyword* category as our objects because the weights of these two categories are the largest and smallest in Table 6. More specifically, we first obtain the corresponding vectors of all tokens in *BasicType* category and *Keyword* category by word2vec [37]. To show the vectors more intuitively, we perform a visualization technique (*i.e.,* TSNE [9]) to visualize them in Figure 7. The figure shows that the vector representations of the token “int” and the token “long” are similar. To obtain more determinate results, we calculate the cosine distance between them. After calculation, we find that their similarity is 87%. For List 1 and List 2, the tokens in *BasicType* category only have token “int” and token “long”, respectively. Therefore, the similarity between the two categories is the similarity between token “int” and token “long”, which is 87%. Such a high similarity makes *CC2Vec* assign a higher weight. Moreover, we also compute the cosine similarity between the tokens “while” and “if”, finding a mere 12% similarity, indicating their dissimilarity. Meanwhile, results in Figure 7 also indicate that the similarity between token “return”, token “while”, and token “if” are low, resulting in the similarity of *Keyword* category between List 1 and List 2 is low. It is the reason why *CC2Vec* assigns a smaller weight to *Keyword* category than *BasicType* category.

In summary, incorporating self-attention mechanism layers in CC2Vec provides interpretability for code clone detection outcomes.

Table 6. Weights of different categories

Token Category	Attention Weight
BasicType	0.1922
Operator	0.1739
Separator	0.1573
Identifier	0.1424
Modifier	0.1424
DecimalInterger	0.1055
Keyword	0.0864

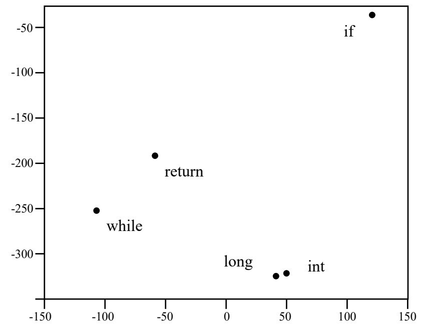


Fig. 7. The visualization of token vectors

4.5 RQ4: Scalability

Our final experiment aims to examine the ability of *CC2Vec* on large-scale code analysis. Specifically, we randomly choose one million code pairs from BigCloneBench and run *CC2Vec* and our comparative systems on them to collect the runtime overheads. We run these tools three times and report the average as their final runtime. For pretrain-based methods (*i.e., Word2Vec*, *Doc2Vec*, *Code2Vec*, *CodeBERT*, *CodeT5*, and *CC2Vec*), the training time indicates the time for pre-trained with the code corpus and the prediction time is the sum of the time to obtain vectors and the time spent in computing the cosine similarity. For example, we should perform contrastive learning to train a self-attention-based program encoder first. Otherwise, we cannot detect code clones by using *CC2Vec*. For deep learning-based methods (*i.e., RtvNN*, *CDLH*, *TBCNN*, *ASTNN*, *SCDetector*, *FCCA* and *CC2Vec-3L*), they need to train a model first, and then the model can be used to detect code clones. In other words, their runtime overheads are divided into two parts, corresponding to training time and prediction time.

The runtime overheads of our proposed methods (*i.e., CC2Vec* and *CC2Vec-3L*) and other detection systems are presented in Table 7. *CC2Vec* is trained faster than other pretrain-based methods. For example, the training time of *Code2Vec* is about 17,345 seconds, and *CC2Vec* takes about 3,649 seconds. The training process for *CodeBERT* is notably resource-intensive and time-consuming [20, 53], hence its training duration is not measured. The prediction time of *CodeT5-base* exceeds that of *CodeBERT*. It is reasonable that *CodeT5* has a greater number of parameters compared to *CodeBERT* and it is also an encoder-decoder model [53]. Since traditional methods do not need to train, their training time is zero. As for prediction time, the token-based method *NIL* consumes the least runtime because it only uses an N-gram representation of token sequences for clone detection. Compared to *NIL*, *CC2Vec* and *CC2Vec-3L* take more runtime to predict the same code

Table 7. Time performance of code clone detectors on analyzing one million code pairs

Tools		Prediction Time	Training Time
Pretrain-based methods	Word2Vec	25s	7,523s
	Doc2Vec	28s	9,126s
	Code2Vec	30s	17,345s
	CodeBERT	274s	-
	CodeT5	873s	-
Traditional methods	SourcererCC	16s	0s
	CCFinder	23s	0s
	NiCad	14s	0s
	Deckard	72s	0s
	CCAligner	10s	0s
	Oreo	15s	0s
	LVMapper	11s	0s
	NIL	9s	0s
	Tamer	10s	0s
Deep learning-based methods	RtvNN	35s	5,206s
	CDLH	90s	45,317s
	TBCNN	86s	41,168s
	ASTNN	2,894s	16,096s
	SCDetector	139s	2,937s
	FCCA	91s	56,769s
Ours	CC2Vec	28s	3,649s
	CC2Vec-3L	83s	4,871s

pairs. It is reasonable because *CC2Vec* conducts lexical analysis to divide the tokens into different categories, which consumes some time to complete the step. And *CC2Vec-3L* fine-tuned based on *CC2Vec* combined with neural networks. Compared to deep learning-based methods, *CC2Vec-3L* are more scalable. It takes about 4,871 seconds to finish the training procedure and 83 seconds to detect one million code pairs. The training phase consumes more time than *SCDetector* (*i.e.,* 2,937 seconds) and more prediction time than *RtvNN* (*i.e.,* 35 seconds), but is faster than other deep learning-based cloned code detection tools. Compared to *RtvNN*, whether from the training or prediction phase, the runtime overhead of *CC2Vec* is lower. For *ASTNN*, it requires 2,894 seconds to predict the one million code pairs, which is about 34 times slower than *CC2Vec-3L*. As for *FCCA*, our method has a significant advantage in detection efficiency during the training phase. In addition, the detecting time of *CC2Vec-3L* is also less than *FCCA* (*i.e.,* the detecting time of *CC2Vec-3L* and *FCCA* are 83s and 91s, respectively).

In summary, due to self-attention mechanism layers and contrastive learning, *CC2Vec* and *CC2Vec-3L* are not as fast as traditional methods. However, it consumes less time than most other pretrain-based methods and deep learning-based methods and also can achieve comparable performance to these deep learning-based methods. Efficient detectors can reduce time and resource consumption and have the potential to detect large-scale code clones.

5 Discussions

5.1 Threats to Validity

The first threat comes from the BigCloneBench dataset, a recent study [31] has shown that the dataset has some quality issues which may cause negative effects on *CC2Vec*. To mitigate the threat and make our experimental results more trustworthy, we also validate *CC2Vec* on another widely used code clone benchmark (*i.e.,* the Google Code Jam dataset). The experimental results still show that *CC2Vec* can perform well on detecting code clones. The second threat comes from the Google Code Jam dataset, it may contain several inaccuracies since some participants may have misunderstood the task and implemented something semantically different. To mitigate the threat, we randomly choose 10% samples from each competition problem and manually verify the functionality. The results show that these samples all correctly address the problem. The third threat comes from the token types, in the token categorization phase, we select a total of 15 categories to commence our experiments. The selection of these categories may cause some inaccuracies since the total number of token categories parsed by *javalang* is not clear. To mitigate the situation, we perform a statistical analysis to select the categories with a high number of occurrences and add a *Null* category to represent the remaining categories. The fourth threat comes from the runtime overhead. The calculation of runtime overheads of *CC2Vec* and its comparative tools may also cause some inaccuracies due to the different machine states such as CPU usage. We mitigate the threat by conducting evaluations three times and reporting the average runtime overhead in our paper.

5.2 Differences From C4

The most similar work to *CC2Vec* is *C4* [48] which uses contrastive learning to detect cross-language code clones. However, our method greatly differs from *C4*. Firstly, *CC2Vec* processes code snippets through syntactic analysis, tokenization, and categorization. Secondly, it utilizes a two-layer attention mechanism to capture the relationships between tokens of the same category and different categories within the language. Finally, we use contrastive learning to derive the code vector. Our primary contribution is how we capture the relationships among tokens in the code, while the subsequent contrastive learning aims to obtain a better code vector. In contrast, *C4* directly tokenizes code snippets and uses CodeBERT for representation, with training via contrastive learning. Contrastive learning is a general method, but our contribution is to capture the relationships between tokens, which is not explored in *C4*. Additionally, our research field differs from that of *C4*. *C4* investigates cross-language clone detection, whereas *CC2Vec* focuses on obtaining better code vectors for more rapid clone detection. We do not use CodeBERT, a large model, but instead train with a lightweight network having only two layers of attention, making our detection more efficient.

5.3 Why does CC2Vec Perform Better?

The reasons why *CC2Vec* is superior to other token-based systems are mainly two-fold. First, *CC2Vec* uses self-attention mechanism layers as the program encoder which can not only compute the corresponding weights to interpret the detection results but also retain some potential relationships between tokens. It is just like a data flow, when some tokens are defined or used earlier, the attention mechanism can establish connections between the definition and the reference locations. These connections can better aid in the detection of clones. In addition, the attention mechanism is a double-direction neural network, which means that it can capture the relationships between tokens better. This method is similar to language in natural language processing, where attention can establish connections between two tokens that are far apart. These potential relationships between tokens can retain some program details between source code tokens. Second, the use of contrastive learning in *CC2Vec* can resist the changes of code structures introduced by different implementations, making it possible to detect some semantic code clones.

6 Related Work

According to the extraction of different representations, existing code clone detection methods can be divided into five main categories: text-based, token-based, tree-based, graph-based, and metrics-based approaches.

Text-based methods [13, 18, 19, 33, 36, 41, 55] compute the similarity between two code fragments in the form of text. Wettel et al. [55] propose a method for clone detection based on code line comparison. Lee et al. [33] design *SDD* to detect large-scale code clones. Token-based methods [21, 26, 34, 39, 44, 52] need to transform the source code into tokens by lexical analysis, then the similarity can be computed by analyzing tokens. Murakami et al. [39] use a hash algorithm to convert the code fragment into a token sequence and introduce a classic and highly efficient Smith-Waterman algorithm to detect code clones. Sainj et al. [44] implement *SourcererCC* to capture the tokens' overlap similarity between different methods to detect near-miss Type-3 clones. *SourcererCC* is the most scalable code clone detector which can scale to scan more than 428 million files on Github [35]. Due to the lack of program details, these text-based and token-based methods are difficult to handle semantic clones.

Tree-based methods [14, 23, 25, 29, 50, 54, 58, 60, 61] need to extract the *abstract syntax tree* (AST) of a code method and then use it to detect code clones. Baxter et al. [14] design a tree-based code clone detector *CloneDR*. It first uses a compiler tool to generate the AST of the code, then calculates the hash code of each subtree, and finally performs similarity matching based on the obtained hash code. Wei et al. [54] introduce *CDLH* which leverages Tree-LSTM [47] on generated binary trees to encode them. Although tree-based methods can detect semantic code clones, however, they suffer from low scalability because of the large execution times [61]. Graph-based methods [16, 28, 30, 45, 51, 59, 62] distill the program semantics into different graph representations, such as *control flow graph* (CFG) and *program dependency graph* (PDG). Komondoor et al. [28] and Krinke et al. [30] both use PDG as the standard to detect code clones. They apply graph analysis to discover isomorphic subgraphs and use these subgraphs to detect semantic code clones. Wu et al. [59] apply centrality analysis to transform the CFG of a method into certain semantic tokens and detect code clones by analyzing these semantic tokens. Graph-based methods can also detect semantic code clones, but due to the complexity of graph isomorphism and the heavy-weight time consumption of graph matching, they can not scan codes on a large scale.

7 Conclusion

In this paper, we propose to boost the capability of code clone systems to efficiently detect syntactic and semantic code clones. To complete the purpose, we introduce *CC2Vec*, a code encoding method for code clone detection. *CC2Vec* uses two self-attention mechanism layers as the program encoder and applies contrastive learning to train the encoder. We can detect simple code clones by directly utilizing *CC2Vec*, while identifying more complicated code clones by combining it with a few neural networks. Through the experimental results, we find that *CC2Vec* can detect more code clones than five pretrain-based methods and 15 state-of-the-art code clone detectors. As for scalability, although *CC2Vec* is not as fast as traditional methods, it surpasses the majority of pretrain-based and deep learning-based methods.

ACKNOWLEDGEMENTS

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-019), and NRF Investigatorship NRF-NRFI06-2020-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] [n. d.]. RMSProp Optimizer. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.
[2] 2017. Google Code Jam. https://code.google.com/codejam/past-contests.
[3] 2020. BigCloneBench. https://github.com/clonebench/BigCloneBench.
[4] 2021. Open Source Technology Trends 2021 – What's there for you? https://www.hiddenbrains.com/blog/open-source-technology-trends.html.
[5] 2021. What the 2021 OSSRA report tells us about the state of open source in commercial software. https://www.synopsys.com/blogs/software-security/open-source-trends-ossra-report/.
[6] 2022. javalang is a pure Python library for working with Java source code. https://github.com/c2nes/javalang/.
[7] 2022. Machine Learning in Python. https://scikit-learn.org/stable/.
[8] 2022. pycparser is a complete parser of the C language. https://pypi.python.org/pypi/pycparser/.
[9] 2022. t-distributed Stochastic Neighbor Embedding. https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html.
[10] 2023. Criticality score. https://github.com/ossf/criticality_score.
[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
[12] Ankesh Anand. 2020. Contrastive Self-Supervised Learning. https://ankeshanand.com/blog/2020/01/26/contrative-self-supervised-learning.html.
[13] Liliane Barbour, Hao Yuan, and Ying Zou. 2010. A technique for just-in-time clone detection in large scale systems. In *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 76–79.
[14] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 368–377.
[15] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591.
[16] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. 175–186.
[17] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Junjie Shan, Caishuang Huang, Wei Shen, Xiaoran Fan, Zhiheng Xi, et al. 2024. StepCoder: Improve Code Generation with Reinforcement Learning from Compiler Feedback. *arXiv preprint arXiv:2402.01391* (2024).
[18] Shihan Dou, Junjie Shan, Haoxiang Jia, Wenhao Deng, Zhiheng Xi, Wei He, Yueming Wu, Tao Gui, Yang Liu, and Xuanjing Huang. 2023. Towards understanding the capability of large language models on code clone detection: a survey. *arXiv preprint arXiv:2308.01191* (2023).
[19] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings of the 1999 International Conference on Software Maintenance (ICSM'99)*. 109–118.
[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[21] Nils Göde and Rainer Koschke. 2009. Incremental clone detection. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (ECSMR'09)*. 219–228.
[22] Tiancheng Hu, Zijing Xu, Yilin Fang, Yueming Wu, Bin Yuan, Deqing Zou, and Hai Jin. 2023. Fine-Grained Code Clone Detection with Block-Based Splitting of Abstract Syntax Tree. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 89–100.
[23] Yutao Hu, Deqing Zou, Junru Peng, Yueming Wu, Junjie Shan, and Hai Jin. 2022. TreeCen: Building tree graph for scalable semantic code clone detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[24] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2020. Fcca: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability* 70, 1 (2020), 304–318.
[25] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. 96–105.
[26] 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* 28, 7 (2002), 654–670.
[27] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *arXiv preprint arXiv:2004.11362* (2020).

[28] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 2001 International Static Analysis Symposium (ISAS'01)*. 40–56.

[29] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*. IEEE, 253–262.

[30] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*. 301–309.

[31] Jens Krinke and Chaiyong Ragkhitwetsagul. 2022. BigCloneBench Considered Harmful for Machine Learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*. IEEE, 1–7.

[32] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[33] Seunghak Lee and Iryoung Jeong. 2005. SDD: high performance code clone detection system for large scale source code. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 140–141.

[34] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CClearner: A deep learning-based clone detection approach. In *Proceedings of the 2017 International Conference on Software Maintenance and Evolution (ICSME'17)*. 249–260.

[35] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages (OOPSLA'17)* (2017), 1–28.

[36] Andrian Marcus and Jonathan I Maletic. 2001. Identification of high-level concept clones in source code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE, 107–114.

[37] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *Computer Science* (2013).

[38] 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*, Vol. 30.

[39] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2013. Gapped code clone detection with lightweight source code analysis. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 93–102.

[40] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. Nil: large-scale detection of large-variance clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–841.

[41] CK. Roy and JR. Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 International Conference on Program Comprehension (ICPC'08)*. 172–181.

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

[43] 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 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*. 354–365.

[44] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, CK. Roy, and CV. Lopes. 2016. SourcererCC: Scaling code clone detection to big code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 1157–1168.

[45] Junjie Shan, Shihan Dou, Yueming Wu, Hairu Wu, and Yang Liu. 2023. Gitor: Scalable Code Clone Detection by Building Global Sample Graph. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 784–795.

[46] Jeffrey Svajlenko, JF. Islam, Iman Keivanloo, CK. Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 International Conference on Software Maintenance and Evolution (ICSME'14)*. 476–480.

[47] Kai Sheng Tai, Richard Socher, and CD. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[48] Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. 2022. C4: Contrastive cross-language code clone detection. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 413–424.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

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

[51] Min Wang, Pengcheng Wang, and Yun Xu. 2017. CCSharp: An efficient three-phase code clone detector using modified pdgs. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*. 100–109.

[52] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and CK. Roy. 2018. CCAligner: A token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 1066–1077.

[53] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[54] 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 2017 International Joint Conferences on Artificial Intelligence (IJCAI'17)*. 3034–3040.

[55] Richard Wettel and Radu Marinescu. 2005. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*. IEEE, 8–10.

[56] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16)*. 87–98.

[57] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K Roy. 2020. Lvmapper: A large-variance clone detector using sequencing alignment approach. *IEEE access* 8 (2020), 27986–27997.

[58] Yueming Wu, Siyue Feng, Deqing Zou, and Hai Jin. 2022. Detecting semantic code clones by building AST-based Markov chains model. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[59] 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 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. 821–833.

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

[61] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. 783–794.

[62] Gang Zhao and Jeff Huang. 2018. Deepsim: Deep learning code functional similarity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*. 141–151.

Received 2023-09-28; accepted 2024-04-16