Machine Learning is All You Need: A Simple Token-based Approach for Effective Code Clone Detection

Siyue Feng*†‡
Huazhong University of Science and Technology, China
fengsiyue@hust.edu.cn

Wenqi Suo*‡
Huazhong University of Science and Technology, China
suowenqi@hust.edu.cn

Yueming Wu§
Nanyang Technological University, Singapore
wuyueming21@gmail.com

Deqing Zou*†‡
Huazhong University of Science and Technology, China
deqingzou@hust.edu.cn

Yang Liu
Nanyang Technological University, Singapore
yangliu@ntu.edu.sg

Hai Jin*¶
Huazhong University of Science and Technology, China
hjin@hust.edu.cn

ABSTRACT

As software engineering advances and the code demand rises, the prevalence of code clones has increased. This phenomenon poses risks like vulnerability propagation, underscoring the growing importance of code clone detection techniques. While numerous code clone detection methods have been proposed, they often fall short in real-world code environments. They either struggle to identify code clones effectively or demand substantial time and computational resources to handle complex clones. This paper introduces a code clone detection method namely *Toma* using tokens and machine learning. Specifically, we extract token type sequences and employ six similarity calculation methods to generate feature vectors. These vectors are then input into a trained machine learning model for classification. To evaluate the effectiveness and scalability of *Toma*, we conduct experiments on the widely used BigCloneBench dataset. Results show that our tool outperforms token-based code clone detectors and most tree-based clone detectors, demonstrating high effectiveness and significant time savings.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

Code Clones, Machine Learning, Token

*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, Cluster and Grid Computing Lab
†Jinyinhu Laboratory, Wuhan, 430074, China
‡School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
§Yueming Wu is the corresponding author
¶School of Computer Science and Technology, HUST, Wuhan, 430074, China

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

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639114

ACM Reference Format:
Siyue Feng, Wenqi Suo, Yueming Wu, Deqing Zou, Yang Liu, and Hai Jin. 2024. Machine Learning is All You Need: A Simple Token-based Approach for Effective Code Clone Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639114

1 INTRODUCTION

Code clone refers to the behaviour of copying and pasting code fragments [8]. Depending on the similarity of the code fragments, code clone is divided into syntactic code clone (*i.e.,* Type-1, Type-2, and Type-3), where two code fragments are syntactically similar, and semantic code clone (*i.e.,* Type-4), where two code fragments use different syntax to achieve the same semantics. Code cloning can be a time and effort saver for software developers, so it is becoming increasingly prevalent as the field of software engineering grows and the demand for code increases. However, code cloning is not an entirely positive endeavour, it can lead to a reduction in code quality [31, 34], the potential for legal conflicts [66], an increase in software maintenance costs [39], and an increased risk of vulnerability propagation [10, 35, 45]. Therefore, the research of code clone detection techniques has become increasingly important.

Nowadays, many code clone detection methods have been proposed. Some tools are dedicated to large scale code clone detection, such as token-based techniques [21, 22, 27, 30, 36, 49, 57], which convert the code into a sequence of tokens to detect clones. Although token-based methods are low-complexity and highly scalable, they cannot detect complex clones because they only consider the lexical aspects of the program and ignore the syntactic information of the code. To address this problem, some tools strive to improve the detection of semantic clones by extracting intermediate representations of the code, *e.g.,* tree-based methods and graph-based methods. Tree-based approaches [28, 29, 37, 60, 67] extract the parse tree of a program (*e.g.,* abstract syntax tree), providing more insight into the syntax of the program. Graph-based approaches [32, 33, 56, 68, 70] extract the graph structure of the code (*e.g.,* program dependency graph and control flow graph), which can contain more semantic details of the code and thus enable more effective detection of semantic clones. Both methodologies have traditionally leveraged tree matching algorithms or graph mining techniques for clone detection. Nonetheless, these methods are accompanied by considerable computational overhead and often lack scalability when

dealing with expansive datasets. In addition to these traditional clone detection methods, a number of learning-based clone detection techniques have emerged in recent years [26, 36, 48, 60, 67, 68]. For example, *ASTNN* [67] is based on the use of recurrent neural network to detect clones, and *SCDetector* [65] employs siamese network for clone detection. In fact, these techniques often apply complex neural networks with underlying techniques and rely on GPUs to train the networks, which are computationally intensive.

Overall, current approaches either have limited ability to identify code clones or incur significant time overhead, or require significant computational resources to cover complex clones. However, a recent study [59] has shown that most code clones in real-world open-source communities are simple clones, and complex clones are rarely seen. Therefore, these approaches that sacrifice scalability to develop complex clone detection capabilities may not be practical in real life and do not enable large-scale code clone detection. Hence, we aim to explore the construction of a simpler, more convenient, and lightweight code clone detection method that can efficiently identify a broader range of complex clones. This pursuit seeks to strike a balance between the effectiveness and the overhead, catering to the demands of clone detection in real-world scenarios.

Specifically, our paper mainly addresses two challenges:

- *Challenge 1: Current methods persistently prioritize the enhancement of the detection of complex semantic code clones at significant costs, even though such clones are infrequent in real-world scenarios. Then, how can we design a lightweight approach to make clone detection tools meet real-life code clone detection needs?*
- *Challenge 2: Training complex neural networks consumes considerable computational resources and is thus very time-consuming during the training phase. Then, how can we avoid high overheads to achieve fast and accurate classification?*

To solve the first problem, we extract the token sequences, which require very little time overhead and can rapidly complete large-scale token extraction. For the extracted token sequences, we use six similarity calculation methods to calculate the similarity of the code pairs. Different similarity calculators can evaluate the degree of code similarity from different perspectives and improve the effectiveness of clone detection. To solve the second problem, we use a machine learning model to process the classification of clones. Machine learning requires only CPU to complete the training process. In comparison to the training of neural networks, training machine learning models can save significant computational resources while further reducing the time overhead and enabling scalability.

We implement a prototype system called *Toma* and compare the effectiveness and scalability of *Toma* with nine code clone detection systems on the widely used *Java* code clone benchmark dataset *BigCloneBench* (BCB) [1, 52]. The baseline system consists of *SourcererCC* [49], *RtvNN* [62] (which are based on token), *Deckard* [28], *ASTNN* [67], *TBCNN* [40], *CDLH* [60] (which are based on tree), *SCDetector* [65], *DeepSim* [68], and *FCCA* [26] (which are based on graph). The results show that *Toma* outperforms detectors based on token in terms of effectiveness by a significant margin, but it is true that *Toma* is not as good as tree-based and graph-based tools in detecting Type-4 clones. However, for the majority of simple code clones occurring in real-world, we are already sufficient for detecting them and can outperform most of the tree-based tools. This phenomenon suggests that the use of machine learning is sufficient to achieve the desired detection results when only code token information is used. Meanwhile, an even greater advantage is that our approach is very scalable and can save time overhead to a large extent. For example, when only using CPU, our method is 65.68 times faster than *DeepSim* in terms of prediction time.

In summary, our contributions to this paper are as follows:

- We present a new lightweight token-based code clone detection method that uses a straightforward approach to extract similarity features of code pairs.
- By training a machine learning model, we implement a prototype system *Toma* [7] to accomplish an effective and scalable code clone detector.
- We analyse the effectiveness and scalability of *Toma* and nine comparison systems on the BigCloneBench [1, 52] datasets. Experimental results show that *Toma* achieves good detection results using only a very short time overhead compared to the comparison systems.
- Our approach demonstrates that machine learning is powerful enough to detect code clones when only simple token feature extraction algorithms are used.

Paper organization. The rest of the paper is organized as follows: Section 2 presents the definition of clone types. Section 3 describes our system. Section 4 reports the evaluation results. Section 5 discusses the paper. Section 6 describes the related work. Section 7 concludes the paper.

2 CLONE TYPES

In our paper, adopting the definitions for each type [11, 47], we categorize code clone into four types based on the degree of similarity:

- **Type-1 (textual similarity)**: Code fragments falling under this type are identical, except for variations in white-space, layout, and comments.
- **Type-2 (lexical similarity)**: Code fragments falling under this type are nearly identical, but they can differ in identifier names, lexical values, as well as white-space, layout, and comments, as seen in Type-1 clones.
- **Type-3 (syntactic similarity)**: Code fragments falling under this type exhibit syntactic similarities but differ at the statement level. In addition to the differences found in Type-1 and Type-2 clones, fragments of this type may have statements added, modified, or removed in comparison to each other.
- **Type-4 (semantically similarity)**: Code fragments belonging to this type are not syntactically similar, yet they implement the same functionality.

To illustrate the difference between these four clone types, Figure 1 gives an example incorporating Type-1 to Type-4 clones. The original fragment computes the factorial of a number. The Type-1 fragment is identical to the original except for the removal of the comment. The Type-2 fragment changes the name of the argument from *n* to *num*. The Type-3 fragment is similar in syntax, but different in statements, changing the method name and argument type, and the condition of the for loop is changed. Finally, the Type-4 fragment uses an iterative method to calculate the factorial of a number, which differs from the original method in syntax and statements but achieves the same functionality.

```
private long factorial(long n){
    long sum = 1;
    for(int i=1;i<=n;i++){
        sum *= i;
    }
    return sum; //return
}// Original
```

```
private long factorial(long n){
    long sum = 1;
    for(int i=1;i<=n;i++){
        sum *= i;
    }
    return sum;
}// Type-1
```

```
private long factorial(long num){
    long sum = 1;
    for(int i=1;i<=num;i++){
        sum *= i;
    }
    return sum;
}// Type-2
```

```
public static int get_factorial(int num){
    int sum = 1;
    for(int i=num;i>=1;i--){
        sum = sum *i;
    }
    return sum;
}// Type-3
```

```
public static int get_factorial(int n){
    if(n == 1){
        return 1;
    }else {
        return n*get_factorial(n-1);
    }
}// Type-4
```

Figure 1: Examples of different clone types

3 SYSTEM

We present the details of our system in this section, including the overall framework and a detailed description of each phase.

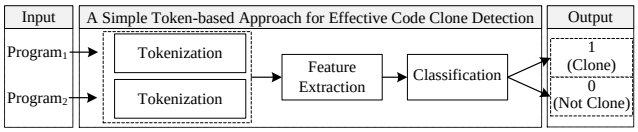

Figure 2: System architecture of *Toma*

3.1 Overview

As depicted in Figure 2, the system consists of three phases: *Tokenization*, *Feature Extraction*, and *Classification*.

- ***Tokenization***: In this phase, the code undergoes lexical analysis, where it is scanned and broken down into individual tokens. The primary input to this phase is the program code, and the resulting output is the token sequence
- ***Feature Extraction***: The main goal of this phase is to extract the similarity features between two token sequences by computing the similarity. The input to this phase is the two token sequences and the output is the similarity features between them.
- ***Classification***: The purpose of this phase is to employ machine learning to predict whether the code pair to be detected is clone or not. The input to this phase is the similarity feature and the output is the classification result of the model.

3.2 Tokenization

In this paper, we want to design a lightweight clone detector using a simple code feature extraction technique, so we take only lexical analysis to extract the code tokens. However, extracting only the tokens may not be able to counteract modifications such as renaming, so we abstract variable names to "V". In addition to this, in order to take the abstraction even further, we convert each extracted token to its corresponding type. For instance, the token "private" can be substituted with its corresponding type "*Modifier*", while the token "num" can be substituted with its type "*Identifier*". Recent work [25, 64] proposes that there are 14 types accounting for over 99.7% of all types, which basically covers most of the tokens. For the tokens that could not be covered, we add an *Null* type to represent them. Therefore, we choose these 15 types as the final token types. This means that all tokens will be classified as one of the 15 types. In this phase, we make the following two operations on the extracted token sequences: i) No processing: We do not modify the token sequence. ii) Token-to-type conversion: We convert the token sequence into a sequence of corresponding token types. The same subsequent operations are performed on these two sequences, and we will implement more detailed experiments in each of the two cases in Section 4.2 to evaluate whether the manipulation of token extraction types improves the detection of the clone detector.

3.3 Feature Extraction

After obtaining the token sequences and type sequences, we measure the similarity of code pairs by calculating the similarity of the sequences of the two code segments. Firstly, we measure similarity by simply counting the number of identical elements between two sequences, which is a common approach for computing similarity. Therefore, we choose two widely used methods to compute similarity: *Jaccard similarity* [43] and *Dice similarity* [55]. However, both of these methods do not take into account any order information, while code, as a form of language, also contains certain grammatical information within the sequence of its tokens. For example, "$D = A + B * C$" and "$D = A * B + C$" are two lines of code sharing entirely identical tokens. However, due to the different order of tokens, they express different syntactic meanings. Therefore, to compensate for the shortcomings of the first two methods in order similarity, we add two similarity calculations *Levenshtein distance* [42] and *Levenshtein ratio* [50]. However, to enhance the detection of Type-3 code clones, we allow tokens to undergo some degree of sequence variation. Therefore, we introduced *Jaro similarity* [20] to detect local similarities, permitting changes in order within a specified matching window. Additionally, the *Jaro-Winkler similarity* [58] adds emphasis on matching common prefixes, and we have also taken this into account. The use of multiple similarity calculation methods allows our method to assess the similarity between two code fragments from different angles (such as common elements, order, local similarity, and prefix similarity) to improve the accuracy of detection. These six algorithms are applied in various fields like information search and data mining [54]. We choose them as our similarity computation approaches for their good performance.

Jaccard similarity coefficient also known as the Jaccard index is a measure commonly used to compare the similarity between sets. The value of *Jaccard similarity* is between 0 and 1, with higher values indicating that the objects being measured are more similar. Given two sets A and B, it is calculated as:

$$Jaccard\,(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{1}$$

Dice similarity coefficient is a set similarity metric, also used to calculate the similarity between two finite sets. The value of *Dice similarity* is between 0 and 1, with higher values indicating that

the objects being measured are more similar. Given two sets A and B, it is calculated as:

$$Dice\,(A,B) = \frac{2 * |A \cap B|}{|A| + |B|} \tag{2}$$

Levenshtein distance is a metric that measures the difference between two sequences. It is a type of Editor Distance and is the measure of the minimum number of editing operations needed to transform one string into another. Permissible editing operations encompass character replacements, insertions, and deletions. The number of edit operations is increased by one if there is an allowed edit operation. The range of *Levenshtein distance* is distributed from 0 to positive infinity, with higher values indicating that the objects being measured are less similar.

Levenshtein ratio is also a measure of the similarity between two sequences. But it is obtained by calculating the edit-like distance. In the edit-like distance, the edit times are increased by one for each occurrence of deletion and insertion of a character, but increased by two for each occurrence of substitution of a character. Given two sequences of strings A and B, and the edit-like distance $ldist$ of them, the *Levenshtein ratio* is calculated as:

$$Levenshtein\_ratio\,(A,B) = \frac{|A| + |B| - ldist\,(A,B)}{|A| + |B|} \tag{3}$$

Jaro similarity measures the similarity of two strings to derive the degree of similarity between them. The value of *Jaro similarity* is between 0 and 1, with higher values indicating that the objects being measured are more similar. For two strings A and B, where $|A|$ and $|B|$ denote the lengths of the strings A and B. m denotes the number of matching characters of the two strings. If the difference between the positions of the same characters in A and B does not exceed the matching window $\left\lfloor \frac{\max(|A|,|B|)}{2} \right\rfloor - 1$, they are considered to be matched. t denotes half of the number of transpositions. The number of transpositions indicates the number of characters in the matching characters that are in different positions. The *Jaro similarity* is given by the following formula:

$$\mathrm{Jaro}\,(A,B) = \frac{1}{3}\left(\frac{m}{|A|} + \frac{m}{|B|} + \frac{m-t}{m}\right) \tag{4}$$

Jaro-Winkler similarity is an algorithm that improves on the *Jaro similarity*. For two strings A and B and their *Jaro similarity* $\mathrm{Jaro}\,(A,B)$, the number of common prefix characters between two strings l, which can be at most 4, a scaling factor constant p that describes the contribution of the common prefix to the similarity, with larger p indicating a greater common prefix weight, up to a maximum of 0.25, taking the default value of 0.1. The *Jaro-Winkler similarity* is given by the following formula:

$$\text{Jaro-Winkler}\,(A,B) = \mathrm{Jaro}\,(A,B) + l * p * (1 - \mathrm{Jaro}\,(A,B)) \tag{5}$$

We calculate six similarity (distance) scores for each pair of token type sequence, and then stitch these six scores into a 6-dimensional vector, which is the extracted similarity feature vector. We will evaluate the different effects and importance of these six features through the interpretability of our method in Section 4.5.

3.4 Classification

In order to address the issue of requiring significant computational resources and practical expenses for training complex neural networks, we utilize machine learning models to handle the classification problem for code clones. Machine learning is widely applied in fields such as data analysis and mining, pattern recognition, and bioinformatics, particularly excelling in handling classification problems. It saves computational resources and requires low time overhead. Therefore, we employ machine learning models to accomplish our classification stage. During the training phase of the model, we extract similarity vectors from the test set. The machine learning algorithm is trained using the input vectors and their corresponding labels. Once trained, the resulting model is saved for future use. During the prediction phase, the code pairs to be tested are also processed in the aforementioned two phases to obtain their feature vectors. These vectors are then inputted into the model for prediction, resulting in the detection outcome for the code pairs (*i.e.,* zero indicating non-clone pairs, and one indicating clone pairs). In Section 4.2, we select different machine learning algorithms such as *k-nearest neighbor* (KNN) [15] and *random forest* (RF) [12] to achieve more comprehensive experiments. Additionally, we also compare the ensemble of all the machine learning algorithms with the use of a single machine learning algorithm to determine whether it can achieve better results in Section 4.3.2.

4 EXPERIMENTS

In this section, we discuss the following five questions through experimental implementation:

- *RQ1: How does the detection performance vary when type are used or not used during the tokenization phase, and different machine learning algorithms are employed during the classification phase?*
- *RQ2: Do individual components have a positive impact on the detection performance?*
- *RQ3: What is the detection performance of Toma compared to other clone detection tools for different types of clones?*
- *RQ4: Which similarity calculation methods are the most effective?*
- *RQ5: What is the time overhead of Toma?*

4.1 Experimental Settings

4.1.1 Experimental Dataset. We conduct our experiments on the widely used *BigCloneBench* (BCB) [1, 52] dataset. The clone pairs from the BCB dataset are at the function level, which aligns with the granularity of our detection method. Furthermore, the clone pairs from the BCB dataset are manually labeled with clone types to facilitate testing the detection effect of *Toma* on different types of clone pairs. In the BCB dataset, the boundary between Type-3 and Type-4 is ambiguous. To address this, Svajlenko et al. [52] subdivided Type-3 and Type-4 into three subtypes based on line-level and token-level similarity: i) *Strongly Type-3* (ST3) clones have a similarity range of [0.7, 1.0), ii) *Moderately Type-3* (MT3) clones have a similarity range of [0.5, 0.7), and iii) *Weakly Type-3/Type-4* (WT3/T4) clones have a similarity range of [0.0, 0.5). Since the BCB dataset only consists of 278,838 non-clone pairs, we randomly select almost 270,000 clone pairs from the total eight million clones. Among them, there are 48,116 T1 clones, 4,234 T2 clones, 21,395 ST3 clones, 86,341 MT3 clones, and 109,914 WT3/T4 clones.

4.1.2 Machine Learning Algorithms. We select some commonly used machine learning algorithms: *k-nearest neighbor* (KNN) [15], *random forest* (RF) [12], *decision tree* (DT) [46], *adaptiva boosting* (Adaboost) [18], *gradient boosting decision tree* (GDBT) [19], *eXtreme Gradient Boosting* (XGBoost) [13], and *logistic regression* (LR) [16] to conduct classification experiments. For KNN, we use 1NN, 3NN, and 5NN, by selecting three widely used K (*i.e.,* K = 1, 3, 5) to commence our experiments. In total, we implement nine machine learning algorithms.

4.1.3 Comparative Tools. There have been numerous code clone detection tools proposed, many of which have demonstrated good performance in terms of effectiveness and scalability. However, most of them are not open source. Therefore, we have chosen the following open-source code clone detection methods to compare with our approach. ***SourcererCC*** [49] : a sophisticated traditional clone detector based on token. ***RtvNN*** [62] : a sophisticated learning-based clone detector that utilizes a *recurrent neural network* (RNN) for its detection mechanism. ***Deckard*** [28] : a sophisticated traditional clone detector based on Abstract Syntax Tree. ***ASTNN*** [67] : a sophisticated learning-based and AST-based clone detector that employs a gate recurrent unit network for its detection process. ***TBCNN*** [40] : a sophisticated learning-based and AST-based clone detector that utilizes a *convolutional neural network* (CNN) for its detection process. ***CDLH*** [60] : a sophisticated learning-based and AST-based clone detector that utilizes a long short-term memory network. ***SCDetector*** [65] : a sophisticated learning-based and CFG-based clone detector that employs centrality and Siamese network. ***DeepSim*** [68] : a sophisticated learning-based and graph-based clone detector that utilizes binary matrix and deep neural network. ***FCCA*** [26] : a sophisticated clone detector that leverages hybrid code representations to achieve its detection capabilities. The parameter settings used in the implementation of these tools are derived from the parameters described in their respective papers that achieved the best results.

4.1.4 Experimental Environment and Metrics. Since we are using a dataset in the *Java* programming language, for the first phase of static analysis, we employ the *Javalang* [5] package in *Python* to tokenize the code. For the second phase of feature extraction, we use the *Levenshtein* [4] package in *Python* to calculate similarity. For the third phase of classification, we utilize *Sklearn* [3] library in *Python* to implement these machine learning algorithms. We conduct all experiments on a server running the Ubuntu 16.04 operating system, with 62GB RAM, an Intel Xeon CPU with 12 cores, and an RTX 2080 Ti GPU. We employ ten-fold cross-validation to record the F1 score, precision, and recall for each validation run, calculate the average of these metrics across the ten validations, and consider this average as the final performance. We use some widely-used metrics for evaluating detection performance. F1 score is defined as $F1 = 2 * P * R/(P + R)$. Precision is defined as $P = TP/(TP + FP)$. Recall is defined as $R = TP/(TP + FN)$. *TP* represents true positive, refers to the number of samples correctly classified as clone pairs, *FP* represents false positive, refers to the number of samples incorrectly classified as clone pairs, and *FN* represents false negative, refers to the number of samples incorrectly classified as non-clone pairs.

4.2 RQ1: Determination of Methods to Be Used at Tokenization and Classification Phases

In the tokenization phase, we not only extract token sequences of program but also transfer these tokens to their corresponding types to get type sequences. In order to verify the effectiveness of these two sequence extraction methods for clone detection, we evaluate the effectiveness in two separate cases in this subsection.

In the classification phase, we employ different machine learning algorithms to classify. We aim to determine which machine learning algorithm yields the optimal detection results for *Toma*.

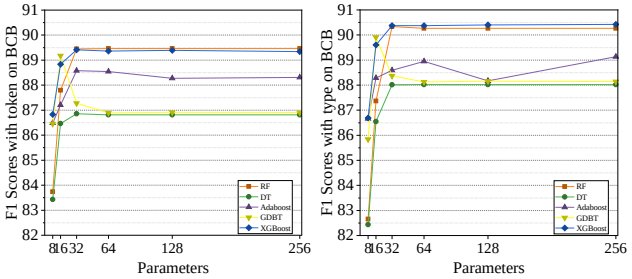

Figure 3: The F1 scores achieved by five machine learning algorithms under different parameter settings

For machine learning algorithms, selecting appropriate parameters is crucial as different parameter choices can significantly impact classification performance. Therefore, we first need to determine suitable parameters for each machine learning algorithm. In the *KNN* algorithm, the neighbor parameter K determines the number of nearest neighbors considered by the model. It is common practice for us to choose odd values such as one, three, or five for K to avoid ties in classification and generally achieve better results in practice. *LR* is a linear classifier that typically does not require explicitly setting parameters like *KNN*. The default values provided in the *Sklearn* package are usually reasonable choices. For *RF*, *DT*, *Adaboost*, *GBDT*, and *XGBoost* algorithms, they are based on tree ensemble methods, they possess a parameter for controlling the depth of the trees. Selecting the depth parameter requires experimentation within a reasonable range. We choose depths of 8, 16, 32, 64, 128, and 256 for these algorithms to test their classification performance and identify the optimal parameter setting.

Figure 3 displays the F1 scores achieved by them under different parameter settings for both token sequences and type sequences. Combining the performance in both cases, we can observe that when the depth parameters for *RF*, *DT*, *Adaboost*, *GBDT*, and *XGBoost* are set to 32, 32, 64, 16, and 32 respectively, they attain their highest F1 scores for the first time. Therefore, we utilize the detection results obtained using these parameter values to compare the performance of different machine learning algorithms.

We plot the F1 values for each machine learning algorithm obtained each time using the ten-fold cross-validation method in Figure 4. From the figure, two phenomena can be observed. Firstly, in comparison to other machine learning algorithms, *GDBT*, *XGBoost*, and *RF* exhibit relatively higher F1 scores. Secondly, using type sequences yields better results compared to token sequences. For instance, *GDBT* achieves an average F1 score of 90.10%, *XGBoost*

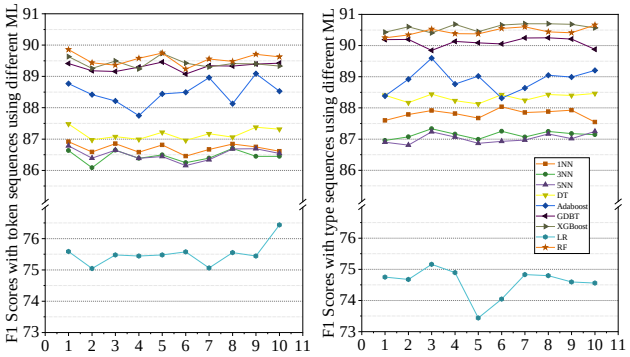

Figure 4: F1 scores of *Toma* using different machine learning classification methods

Table 1: The P-values between each pair of algorithms, with ● indicating a P-value less than 0.05 and ○ indicating a P-value greater than 0.05

		P-value						Mean(%)
		token			type			
		G	X	RF	G	X	RF	
token	G	○	○	●	●	●	●	89.31
	X	○	○	○	●	●	●	89.46
	RF	●	○	○	●	●	●	89.56
type	G	●	●	●	○	●	●	90.10
	X	●	●	●	●	○	○	90.59
	RF	●	●	●	●	○	○	90.45

achieves 90.59%, while *RF* achieves 90.45% when type sequences are used. In contrast, *1NN* achieves an average F1 score of 87.80%, *3NN* achieves 87.14%, *5NN* achieves 87.02%, *DT* achieves 88.33%, *Adaboost* achieves 88.89%, and *LR* achieves 74.57%. When token sequences are used, *GDBT*, *XGBoost*, and *RF* achieve an average F1 score of 89.31%, 89.46%, and 89.56%, respectively, which are also better than other machine learning algorithms. However, the top-performing three machine learning algorithms exhibit minimal differences between each other (0.2% on average). Furthermore, compared to the case of using token sequences, the average improvement of all three results is about 1%.

Therefore, we conduct a statistical test on the results of *GDBT*, *XGBoost*, and *RF* in both cases to validate whether there are significant differences between the results. Due to the non-normal distribution of our data, we used the *Kolmogorov-Smirnov Test* (ks-test) [38] for statistical testing. The ks-test is a common non-parametric test, and the two-sample test is used to determine whether two samples belong to the same distribution. When the calculated P-value [61] is less than 0.05, it indicates that the two corresponding algorithms have significantly different effects. Table 1 displays the P-values between each pair of algorithms, where G denotes *GDBT* and X denotes *XGBoost*. Since this matrix is a symmetric matrix, we can just focus on the upper triangular region. We need to select the optimal combination from three machine learning algorithms and two sequences. The selection follows these principles: when there is a significant difference in performance, we choose the method with better performance, and when there is no significant difference, we choose the more scalable approach.

From the table, we can observe that there is a significant difference in the effect of using type sequences and token sequences, regardless of the machine learning algorithm (as shown by the nine ● with yellow backgrounds in the table). The performance of using type sequences is higher than that of token sequences. These results indicate that incorporating type information can improve the detection effectiveness in code clone detection. By extracting token types alongside the tokens themselves, we reduce false negatives caused by local variations, thereby enhancing the recall of clone pairs. Therefore, type sequences will be used in the subsequent comparative experiments.

We continue to observe the performance of three machine learning algorithms in the case of using type sequences. Among them, *GDBT* has a significant difference from *RF* and *XGBoost* (as shown by the two ● in the green background in the table), *GDBT* has worse performance, so it is excluded. As for *RF* and *XGBoost*, there is no significant difference in the performance between them (as shown by the ○ on the red background in the table). However, the training speed of *RF* is about 3.5 times faster compared to *XGBoost*. Therefore, in the subsequent comparative experiments, we select *RF* to compare its classification performance.

> ***In Summary:*** *When type sequence is used during the tokenization phase, and RF is employed during the classification phase, Toma can have a relatively high F1 score and is more scalable. As a result, we select type sequences and RF as the components of Toma.*

4.3 RQ2: Individual Components Effectiveness

4.3.1 Machine Learning. To investigate the improvement of machine learning in code clone detection, we conduct this experiment. We employ these similarity calculation methods directly to assess the similarity of type sequences without utilizing machine learning algorithms. If the similarity score for a pair of codes surpasses a specified threshold, they will be labeled as a clone. Our experiments record the clone detection performance using six similarity calculation methods under various thresholds. Then we compare these results with the performance achieved using the *RF* algorithm.

Table 2: The detection performance of each similarity calculation method and random forest

Metrics	P	R	F1	P	R	F1	P	R	F1
Threshold	150			100			50		
L_dis	53.1	80.6	**64.0**	54.9	73.0	62.7	63.4	56.9	60.0
Threshold	0.7			0.8			0.9		
Jaccard	80.7	63.3	**71.0**	90.2	48.3	63.0	99.0	29.1	45.0
Dice	70.0	76.7	**73.2**	77.9	66.9	71.9	92.1	44.8	60.3
Jaro	62.6	83.6	**71.6**	80.5	63.7	71.1	99.6	26.2	41.5
JW	62.6	**83.6**	71.6	74.1	73.0	**73.5**	96.5	38.7	55.2
L_ratio	76.0	70.8	**73.3**	95.3	47.4	63.3	99.8	27.8	43.5
RF	Precison = **92.9**, Recall = **87.9**, F1 score = **90.3**								

The experimental results are presented in Table 2, with *JW* represents *Jaro_winkler*, *L_dis* represents *Levenshtein_distance*, and

L_ratio represents *Levenshtein_ratio*. *Levenshtein_distance* is a distance calculation method, so its threshold is different from other similarity calculation methods. For instance, when the threshold is set to 0.8, using the *Jaro_winkler* similarity calculation method achieves the highest F1 score of 73.50% among all similarity methods. At this time, its precision is 74.07%, and recall is 72.95%. In contrast, *RF* achieves 92.95% precision, 87.87% recall, and 90.34% F1 score. Compared to using only a specific similarity calculation method, the machine learning algorithm significantly enhances the code clone detection performance. This improvement is attributed to the machine learning algorithm's ability to comprehensively utilize the outputs of different similarity calculation methods and automatically assign weights to similarity scores from different perspectives. Consequently, it can consider code clone similarity more comprehensively, mitigating the bias caused by a single method, and thus achieving better detection results. The use of multiple similarity calculation methods allows our method to assess the similarity between two code fragments from different angles, thereby improving the accuracy of detection.

4.3.2 Ensemble Learning. The machine learning algorithms we have chosen excel in solving classification problems, employing different algorithmic logic to classify the same problem. In the previous subsection, we discuss the comparative performance of these machine learning algorithms within our method. However, their specific performance depends on the nature of the classification task and the characteristics of the data. Each machine learning algorithm has its own strengths and advantages. Given the diverse advantages of different machine learning algorithms, we consider the utilization of ensemble learning to integrate the results from multiple classifiers and investigate whether their combination could lead to better performance in terms of accuracy and robustness for our clone detection tool.

Table 3: The detection performance of ensemble learning and random forest, and the P-value between these scores

	F1	Precision	Recall
Random Forest	90.39%	92.98%	87.95%
Ensemble Learning	90.25%	93.27%	87.42%
P-value	○	○	●

We construct a meta-classifier that combines the predictions of multiple base classifiers. These base classifiers are trained using the nine machine learning algorithms mentioned in RQ1, and each provides predictions for the given code pairs to be detected. The meta-classifier learns to make the final prediction based on the outputs of the base classifiers, effectively leveraging their diverse perspectives. In the ensemble learning approach, if the number of base classifiers predicting "clone" for a code pair is equal to or greater than five, the ensemble learning classifier determines the code pair as a "clone". Otherwise, it is classified as a "non-clone".

To evaluate the effectiveness of ensemble learning, we compare the performance of the individual classifier (*i.e., RF*) selected in RQ1 with the ensemble learning classifier. From Table 3, we can observe that the F1 scores and Precision of ensemble learning are not significantly different from those of *RF*, and the F1 score is even slightly lower than those of *RF* in terms of the average value. Furthermore, the Recall is significantly lower than *RF*. This may be due to the fact that ensemble learning classifier identifies code pairs as "clone" only when more than or equal to five classifiers give "clone" predictions. Therefore, ensemble learning has a lower recall. In addition, some of the selected machine learning methods may not be suitable for our approach, introducing noise to the detection results and interfering with the final decision of the ensemble learning. As a result, the effectiveness of ensemble learning is not as outstanding as we initially anticipated.

In Summary: *Compared to using only a specific similarity calculation method, the machine learning algorithm significantly enhances the detection performance. The effectiveness of ensemble learning is not as outstanding as we initially anticipated, so we do not use it.*

4.4 RQ3: Overall Effectiveness

In the previous experiments, we have already demonstrated the accuracy of using token types and the *RF* algorithm in code clone detection. Now, we compare the detection results by using these two approaches with other comparative tools to further analyze our overall effectiveness. Firstly, we run *Toma* on the BCB dataset that contains clone pairs of all types and compare *Toma* against baseline techniques to evaluate its effectiveness in detecting all types of clones. Next, we specifically evaluate the performance of *Toma* on different types of clone pairs, such as T1, T2, ST3, MT3, and WT3/T4. We compare the performance of *Toma* with baseline tools to assess its effectiveness in detecting different types of clones.

Table 4: Results on BigCloneBench dataset

Group	Method	F1	Precision	Recall
Token-based	SourcererCC	0.14	**0.98**	0.07
	RtvNN	0.01	0.95	0.01
Tree-based	Deckard	0.12	0.93	0.06
	ASTNN	0.93	0.92	0.94
	TBCNN	0.85	0.90	0.81
	CDLH	0.82	0.92	0.74
Graph-based	SCDetector	0.92	0.97	0.94
	DeepSim	**0.98**	0.97	**0.98**
	FCCA	0.92	**0.98**	0.95
Our method	Toma	0.90	0.93	0.88

Table 4 provides a comprehensive comparison of the performance of *Toma* with other tools. As a token-based method utilizing machine learning, our approach effectively balances precision and recall, resulting in F1 scores that surpass the majority of the token-based and tree-based compared tools, but not as good as graph-based comparison tools.

Compared to token-based tools, our F1 score significantly surpasses the two compared tools. This can be attributed that *SourcererCC* is a token-based clone detection tool that primarily considers token overlap similarity to detect code clones. While *SourcererCC* performs well in terms of clone detection accuracy, its limited ability to handle complex clone code arises from the lack of consideration for semantic information and the absence of machine learning techniques. This limitation leads to lower F1 scores. *RtvNN* is also a

token-based tool that uses recursive neural networks to compute the similarity of code pairs based on the Euclidean distance between tokens and abstract syntax trees. However, this distance calculation approach results in minimal differences between most code pairs, making it difficult to effectively differentiate between clones and non-clones. *RtvNN* relies on threshold settings to maintain high precision but cannot achieve a balance between precision and recall, resulting in very low F1 scores.

Compared to tree-based tools, *Toma*'s detection performance outperforms the majority of them. *Deckard* exhibits low recall but high precision. This is because *Deckard* represents the syntax structure information of the parse tree using vectors and then uses clustering to find neighboring vectors to determine the similarity of code pairs. This approach requires the feature vectors of parse tree roots to be very close. As a result, *Deckard* achieves high precision when detecting highly similar clone pairs (*i.e.,* T1 or T2) in our dataset. However, in slightly more complex code clone pairs, where more than half of them have different parse tree structures, *Deckard* fails to recognize these clones, resulting in low recall and consequently low F1 scores. *ASTNN* decomposes each large AST into a series of smaller statement trees and encodes them as vectors. It utilizes BiGRU and RvNN encoders to automatically capture more important node information, resulting in the highest F1 score. The detection outcomes of *TBCNN* and *CDLH* exhibit a slightly reduced performance compared to *Toma. CDLH* undertakes an initial transformation of program ASTs into binary trees, subsequently utilizing binary Tree-LSTM [53] for tree representation. However, this binary tree conversion raises concerns related to the long-term dependency of the original semantics, thereby impeding the achievement of optimal detection results. Similarly, the utilization of a sliding convolution kernel in *TBCNN* gives rise to challenges related to the absence of context information over extended distances, potentially leading to suboptimal detection outcomes, especially as the depth of the AST increases.

Compared to graph-based tools, all of them achieve improved detection results by leveraging semantic information. *SCDetector* follows a different approach by associating semantic details with tokens based on the analysis of each basic block's centrality in the control flow graph. *DeepSim* adopts a strategy where it abstracts variables and basic code blocks, along with their relationships, into a binary matrix. This matrix is then utilized as input for a deep learning model, enabling effective detection of semantic clones and yielding ideal performance in this regard. *FCCA* stands out by extracting diverse representations of code, including token, tree, and graph. These representations are processed by deep learning model that incorporates an attention mechanism, enabling the detection of most semantic code clones.

In contrast, our method only extracts lexical information from the code and does not incorporate additional structural or semantic information. The absence of additional code information in our method leads to the inability to detect more clone pairs, resulting in lower recall. However, while most tree-based and graph-based methods achieve good detection performance by extracting semantic information from tree and graph using neural networks, with significant time and computational resource overhead. They require the involvement of GPUs for complex network training, whereas our method can be completed using CPU alone. Therefore, *Toma* is more scalable compared to tree-based and graph-based methods.

Table 5: The F1 score, Precision, and Recall in detecting each clone type

Metrics	T1	T2	ST3	MT3	WT3/T4
F1	1.00	1.00	0.99	0.93	0.77
Precision	1.00	1.00	0.99	0.94	0.86
Recall	1.00	1.00	0.99	0.93	0.70

Next, we proceed with an analysis of *Toma*'s performance in detecting different types of clones. The F1 scores obtained for each type will be compared with those of the baseline tools. As shown in Table 5, *Toma* achieves a precision, recall, and F1 score of 99% when detecting clones of types T1, T2, and ST3. When detecting clones of type MT3, *Toma* maintains a relatively high recall score of 93%. This suggests that *Toma* is sufficient to detect most clones in the real world. When it comes to detecting clones of type WT3/T4, *Toma*'s performance is relatively weaker. This is because clones of these types often exhibit higher complexity and variability. However, *Toma* solely considers the lexical information of the code, and the token sequences only reflect a very basic code structure. Despite of these limitations, our method still achieves an F1 score of 77%, providing strong evidence that a simple approach based on machine learning algorithms is sufficient for detecting the majority of clones.

Table 6: F1 scores for each clone type of *Toma* and comparative tools

Group	Method	T1	T2	ST3	MT3	WT3
Token-based	SourcererCC	1.00	1.00	0.65	0.20	0.02
	RtvNN	1.00	0.97	0.6	0.03	0.00
Tree-based	Deckard	0.73	0.71	0.54	0.21	0.02
	ASTNN	1.00	1.00	0.99	**0.98**	0.92
	TBCNN	1.00	1.00	0.93	0.80	0.86
	CDLH	1.00	1.00	0.94	0.88	0.82
Graph-based	SCDetector	1.00	1..00	0.97	0.97	0.94
	DeepSim	0.99	0.99	0.99	**0.98**	**0.95**
	FCCA	1.00	1.00	0.99	0.97	**0.95**
Our method	Toma	**1.00**	**1.00**	**0.99**	0.93	0.77

Table 6 presents the F1 scores of the baseline tools and *Toma* in detecting different clone types. It can be observed that *Toma* outperforms the majority of the baseline tools when detecting clones of types T1, T2, and ST3. In the case of detecting clones of type MT3, *Toma* achieves an F1 score of 93%, slightly lower than *ASTNN*'s F1 score of 98% and graph-based tools, but with a significant advantage over the other tree-based tools and token-based tools. When detecting clones of type WT3/T4, *SourcererCC*, *RtvNN*, *Deckard*, *ASTNN*, *TBCNN*, *CDLH*, *SCDetector*, *DeepSim*, and *FCCA* can attain the F1 scores of 2%, 0%, 2%, 92%, 86%, 82%, 94%, 95%, and 95% respectively. *Toma* achieves an F1 score of 77%. It is evident that our method performs significantly better than token-based tools, but falls short compared to tools based on graph and tree, except for *Deckard*. This is because tree-based methods consider the syntactic information of the code, which can reflect the similarity between code pairs at the syntactic level. Graph-based methods consider more semantic information such as control flow and data flow of the code. Our

approach only extracts the lexical information of the code and does not extract any syntactic and semantic information, thus lacking of the ability to detect semantic clones. Through the current recent research [59], we can clearly know that most of the real-life clones are simple clones (*i.e.,* T1, T2, ST3, and MT3), and the proportion of complex semantic clones is very small (*i.e.,* 6%). Therefore, the effectiveness of *Toma* is sufficient to detect most of the clones in the real world and can meet the needs of real-life.

> ***In Summary:*** *Overall, although Toma is not as effective as tree-based and graph-based methods in detecting complex clones, it has been able to achieve high enough effectiveness in detecting some simple clones and can meet the needs of real-life.*

4.5 RQ4: Interpretability

In order to ascertain the most effective similarity calculation methods among the six used in our approach, we leverage the interpretability of machine learning algorithms to assess the importance of each feature. By extracting the weights assigned to each feature in the input feature vectors of the random forest algorithm, we can gain a clear understanding of the significance of these six similarity calculation methods for clone detection.

From Table 7, we can observe that the feature derived from the *Levenshtein_distance* calculation exhibits the highest level of importance, followed by the *Levenshtein_ratio* feature. This is because the *Levenshtein_distance* represents the minimum number of edit operations required to transform one sequence into another, and this count of edits directly reflects the differences between the two sequences. Furthermore, the *Levenshtein_distance* partially preserves the order of tokens in the sequence, which enables it to capture modifications involving statement insertions and deletions, thus enhancing the precision of *Toma* in detecting Type-3 clones. In contrast, the *Levenshtein_ratio* incorporates the lengths of the two sequences, allowing for differentiation between sequences of varying lengths even when they have the same class of edit distance. In the context of clone detection, we may prioritize capturing the differences between code fragments rather than their lengths. Therefore, the introduction of sequence lengths by the *Levenshtein_ratio* does not confer higher importance for clone detection purposes.

Table 7: The importance of features in detecting code clones

Rank	Feature Name	Weight
1	Levenshtein_distance	0.241239
2	Levenshtein_ratio	0.226567
3	Jaro_winkler	0.172471
4	Jaro	0.134083
5	Dice	0.11441
6	Jaccard	0.11123

Both *Jaro_winkler* and *Jaro* similarity metrics incorporate a threshold matching window. This setting emphasizes local similarity and can accommodate certain changes in the order of statements while limiting code variations within a certain range. The emphasis on local similarity judgments enhances detection accuracy but sacrifices recall. The *Jaro_winkler* algorithm places greater emphasis on the importance of matching prefixes compared to *Jaro* similarity.

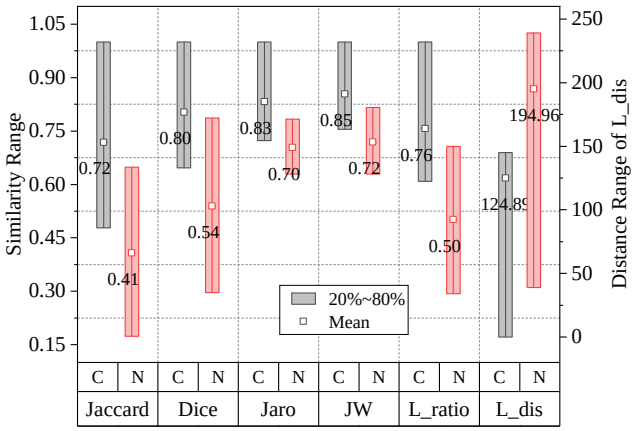

Figure 5: The distribution of similarity scores (distances) computed by each method

If two strings have the same initial characters, they will receive a higher similarity score. This algorithm aligns more closely with the development patterns observed in code, such as control flow and data flow. It assumes that two code fragments with the same semantic meaning have the same semantic understanding at the beginning of the code. Therefore, the *Jaro_winkler* algorithm is considered to have higher importance in clone detection tasks.

Two similarity algorithms that have relatively lower importance are *Dice* and *Jaccard*. These two similarity calculation methods focus only on the number of shared features between sequences, while disregarding the order of the sequences. However, the execution order of identical statements can influence the semantic meaning of a program, the order of token sequences largely reflects the progression of the program and its control or data flow changes. The disregard for sequence order in *Dice* and *Jaccard* can lead to a certain amount of false positives, hence their lower importance.

To assess the efficacy of similarity calculation methods in discerning between clone and non-clone pairs, we analyze the distribution of computed similarity scores or distances for each method. The results are recorded in Figure 5, where *JW* represents *Jaro_winkler*, *L_ratio* represents *Levenshtein_ratio*, and *L_dis* represents *Levenshtein_distance*. *Levenshtein_distance* is a distance calculation method, a smaller distance score indicates higher similarity, while a larger score suggests less similarity. The data in the figure clearly demonstrates that there is a distinct difference in the distribution of similarity scores between clone pairs and non-clone pairs, indicating that those similarity calculation methods are effective in distinguishing clones. To demonstrate the significance of their differences, we also calculate the P-value of the similarity scores between clone and non-clone pairs for these similarity calculation methods. The results show that the P-value for all six sets of data is much less than 0.05, for example, the P-value for *Levenshtein_distance* is $1.72e^{-49}$. Furthermore, different similarity calculation methods have varying levels of discriminative power. For instance, the *Levenshtein_distance* method shows considerable overlap in the distribution of distance scores between clone and non-clone pairs. This may be because the distance scores span from zero to positive infinity, where scores

tend to increase with longer sequences. This complexity hinders effective detection using a straightforward threshold-based method. With the incorporation of machine learning, the sequence information in *Levenshtein_distance* can be leveraged effectively, becoming the most important feature in the detection process.

In Summary: *Levenshtein_distance is the most important feature, its count of edits directly reflects the differences between the two sequences. The observed score distribution disparity between clone pairs and non-clone pairs suggests the efficacy of the employed similarity calculation methods in distinguishing clones.*

4.6 RQ5: Scalability

In this section, we analyze the scalability of our system by comparing the runtime overhead of *Toma* against baseline tools. We randomly select 1,000,000 pairs of code from the BCB dataset as test set. We perform ten random selections, resulting in ten sets of test objects. The average runtime overhead of each set is calculated and presented as the final result. To give a more comprehensive comparison of these code clone detection methods in real-life scenarios, we also run these algorithms in an environment with only CPU, to observe how their training time and prediction time would change without GPU acceleration.

It can be observed that compared to non-learning-based clone detection tools, *Toma* demonstrates a prediction time that is only slightly longer than that of *SourcererCC*, with a mere 4-second disparity. This is because *SourcererCC* uses overlap similarity based on *Jaccard* index, while *Toma* uses six similarity calculation methods, including *Jaccard* similarity, which requires more time for processing. *Deckard* has a slightly longer prediction time than *Toma*, but its overall runtime is shorter because the tree-based clone detection tool *Deckard* extracts the structural information of syntax trees into vector representations and determines clones through vector clustering, resulting in a short prediction time.

Table 8: Time performance of the baseline tool and *Toma* with and without GPU, where T denotes training time and P denotes prediction time

Group	Method	GPU		CPU	
		T(s)	P(s)	T(s)	P(s)
Token-based	SourcererCC	-	-	0	12
	RtvNN	3,862	26	23,176	1,747
Tree-based	Deckard	-	-	0	53
	ASTNN	11,940	2,147	101,833	4,851
	TBCNN	30,538	64	201,167	1,914
	CDLH	33,616	67	219,436	2,020
Graph-based	SCDetector	2,179	103	17,640	2,985
	DeepSim	10,048	25	62,903	1,445
	FCCA	42,111	68	243,621	2,221
Our method	Toma	-	-	914	22

Compared to learning-based approaches, although both training and prediction processes of these methods are accelerated using GPUs, *Toma* still requires the least amount of training and prediction time. When testing the runtime of these tools using only the CPU, we obtain significantly slower results. For example, in terms of prediction time (which is more important than training time), the fastest *DeepSim* takes 1,445 seconds to complete its prediction, which is 65.68 times longer than *Toma*. This proves once again that these approaches that sacrifice scalability to develop complex clone detection capabilities may not be practical in real life. This is because deep learning and neural network training itself require significant computational resources and consume a considerable amount of time. In contrast, *Toma* leverages token sequences acquired through lexical analysis, employs straightforward similarity techniques to extract features, and utilizes a lightweight training process for the random forest machine learning algorithm. This training process only demands CPU resources and is time-efficient. Therefore, the prediction process exhibits notable speed.

In Summary: *Toma achieves a good balance between detection performance and runtime efficiency, providing efficient clone detection with minimal time overhead.*

5 DISCUSSION

5.1 Threats to Validity

5.1.1 External Validity. The first threat comes from the number of types. In the tokenization phase, we transform the extracted token sequences into type sequences. 99.7% tokens can be categorised into 14 types, but these 14 types may not be enough to cover all tokens. To mitigate this threat, we add a *Null* type to cover the tokens beyond the 14 types. This addition helps to improve the generality of our methods by extending them to more *Java* codes. The second threat stems from our dataset. We only evaluated our method on the BCB dataset, which can not be representative of all data. Therefore, we plan to use more challenging and more representative datasets to further evaluate the capabilities of our method in future work.

5.1.2 Internal Validity. The first threat arises from the selection of parameters in the machine learning model, as diverse parameter choices can lead to outcome variations. To mitigate this threat, we evaluate the detection performance of different depth parameter values on our dataset, aiming to determine the optimal value for the machine learning methods. When the depth parameters for *RF*, *DT*, *Adaboost*, *GBDT*, and *XGBOOST* are respectively set to 32, 64, 256, 16, and 256, they achieve the best F1 scores on our dataset for the first time. The second threat arises from the recording of time overhead. Different machine states, such as CPU utilization, may impact the running time of the experiments. So, it is hard to obtain absolutely accurate and universally applicable data. To mitigate this threat, we conduct ten experiments and record the time overhead for each run. By calculating the average time overhead from multiple runs, we can reduce the influence of individual experimental variations and provide more reliable and stable time overhead data.

5.2 Programming Language Generalizability

In our paper, we design our tool specifically for *Java* since the test dataset is based on a *Java* corpus. The subsequent stages of clone detection are language-independent and do not require modifications. For example, utilizing dedicated tools like *pycparser* [6] designed for lexical analysis in *C* language can enhance our approach to code clone detection in *C*.

5.3 Colinearity of Features

We conduct an analysis of the collinearity [17] of features. Collinearity among features can pose issues for the interpretability of feature weights in linear regression models. However, for tree models like *RF* used in our experiment, it usually does not have a significant impact on the predictive performance and interpretability even if there is collinearity among features. Nonetheless, we still use the Pearson correlation coefficient [14] to calculate the correlation between our features. The Pearson correlation coefficient ranges from -1 to 1, and the closer to zero the value is, the weaker the correlation between features. The experimental results show that the absolute values of the correlation coefficients between our features are far less than 0.01, indicating that our features operate from different perspectives. Due to space constraints, the complete experimental results are presented on our website [7].

6 RELATED WORK

In this section, we will focus on current research related to code clone detection. Efforts in this field can be categorized into two main aspects: improving scalability and enhancing the effectiveness of semantic clone detection.

Efforts to improve scalability in clone detection typically involve token-based clone detection techniques [21, 22, 27, 30, 36, 41, 49, 57]. These techniques extract token sequences from the target source code. Subsequently, they compare the token sequences' similarity to detect code clones by searching for duplicate token subsequences. Golubev et al. [22] proposes a modification of the token bag-based clone detection technique by implementing a multi-threshold search to detect more clones, to obtain a larger diversity of pairs without loss of precision. *NIL* [41] utilizes N-grams to construct a reverse index from the extracted token sequences, and subsequently employs the common subsequence technique to verify clones. This approach demonstrates excellent performance in detecting T1 and T2 type clones, as well as some highly similar T3 clones. Moreover, it exhibits strong scalability. An optimization method is also proposed that significantly reduces the overlap of detected clones between searches and can detect T3 clones.

In order to enhance the capability of code clone detection for semantic clones, some methods focus on extracting representations of code trees [23–25, 28, 29, 37, 44, 60, 63, 64, 67, 69] and graphs [32, 33, 51, 56, 65, 68, 70], and utilize machine learning and deep learning [36, 60, 65, 67, 68] to improve the detection of semantic clones. *CClearner* [36] is the first token-based clone detector to employ deep learning. It detects clones by training deep learning models with token sequences as input. *CDLH* [60] is a clone detection tool based on AST, using LSTM networks to extract binary tree representations as vectors. *ASTNN* [67] utilizes a bidirectional RNN model to integrate subtree vectors predefined by rules into the final vector representation. It achieves high precision in detecting semantic clones but lacks scalability. *DeepSim* [68] leverages PDG-based semantic high-dimensional sparse matrices to extract feature vectors and then combines deep learning models for clone detection. *SCDetector* [65] is based on control flow graphs, utilizing social network centrality and Siamese neural networks. The tree-based and graph-based methods extract rich semantic information from intermediate representations of code, and exhibit strong capabilities in detecting complex clones. Learning-based code clone detection techniques with powerful learning abilities can effectively extract semantic information from code, thereby improving the detection of semantic clones.

In summary, traditional token-based methods are only able to detect very simple lexically similar clones and lack the ability to deal with some slight modifications in the statements. Tree-based and graph-based approaches tend to prioritize the extraction of semantically rich intermediate representations to achieve high-precision detection of complex semantic code clones. However complex matching algorithms impose high time overheads. Learning-based approaches typically involve the deployment of intricate neural network architectures that consume a lot of computational resources. Compared with these methods, our method only extracts the type sequences through lexical analysis and employs six similarity calculation methods to increase the representation of the code from different perspectives. Subsequently, the use of machine learning models makes our approach accurate, scalable, and interpretable without the need for large computational resources.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose a code clone detection method based on tokens and machine learning. We start by extracting token sequences of code through lexical analysis. Then six similarity calculation methods are used to obtain six similarity scores between the two type sequences, forming a feature vector. This vector is then fed into a trained machine learning model to determine whether the code pairs are clones. We conduct effectiveness and scalability comparisons of our system with nine code clone detection systems on the widely used dataset BigCloneBench. Experimental results show that our method can ensure detection performance surpassing most tree-based clone detectors with minimal time overhead. Besides, our method contributes to exploring the powerful detection capabilities of machine learning. We demonstrate that even with simple feature extraction algorithms, machine learning still possesses strong clone detection capabilities. In fact, with generative AI [9], such as ChatGPT [2], it is easier than before to transform a code segment into another with the same semantics but with different syntax. This may increase the ratio of complex code clones in our real world. In our future work, we will explore more machine learning methods, more token details, and some lightweight tree details to enhance the ability to detect such complex clones.

ACKNOWLEDGEMENTS

This work is supported by the Key Program of National Science Foundation of China under Grant No. 62172168 and is partially 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] 2022. BigCloneBench. https://github.com/clonebench/BigCloneBench.
[2] 2023. ChatGPT: A free-to-use AI system. https://chat.openai.com/.
[3] 2023. An open source machine learning library that supports supervised and unsupervised learning. (Scikit-learn). https://scikit-learn.org/stable/.
[4] 2023. A pure Python library for fast computation of various distances. https://github.com/ztane/python-Levenshtein/.
[5] 2023. A pure Python library for working with Java source code, provies a lexer and parser targeting Java 8. (Javalang). https://pypi.org/project/javalang/.
[6] 2023. Pycparser is a complete parser of the C language. https://pypi.python.org/pypi/pycparser/.
[7] 2024. The source code and data of Toma. https://github.com/CGCL-codes/Toma/.
[8] Qurat Ul. Ain, Wasi H. Butt, Muhammad W. Anwar, Farooque Azam, and Bilal Maqbool. 2019. A systematic review on code clone detection. *IEEE Access* 7 (2019), 86121–86144.
[9] David Baidoo-Anu and Leticia Owusu Ansah. 2023. Education in the era of generative artificial intelligence (AI): Understanding the potential benefits of ChatGPT in promoting teaching and learning. *Journal of AI* 7, 1 (2023), 52–62.
[10] Brenda S. Baker. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE'95)*. 86–95.
[11] 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.
[12] Leo Breiman. 1996. Bagging predictors. *Machine Learning* 24, 2 (1996), 123–140.
[13] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. 785–794.
[14] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise Reduction in Speech Processing* (2009), 1–4.
[15] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* 13, 1 (1967), 21–27.
[16] David R. Cox. 1958. The regression analysis of binary sequences. *Royal Statistical Society: Series B (Methodological)* 20, 2 (1958), 215–232.
[17] Donald E. Farrar and Robert R. Glauber. 1967. Multicollinearity in regression analysis: the problem revisited. *The Review of Economic and Statistics* (1967), 92–107.
[18] Yoav Freund and Robert E. Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Computer and System Sciences* 55, 1 (1997), 119–139.
[19] Jerome H. Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of Statistics* (2001), 1189–1232.
[20] Najlah Gali, Radu Mariescu-Istodor, Damien Hostettler, and Pasi Fränti. 2019. Framework for syntactic string similarity measures. *Expert Systems with Applications* 129 (2019), 169–185.
[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] Yaroslav Golubev, Viktor Poletansky, Nikita Povarov, and Timofey Bryksin. 2021. Multi-threshold token-based code clone detection. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*. 496–500.
[23] 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 (ISSTA'23)*. 89–100.
[24] Yutao Hu, Yilin Fang, Yifan Sun, Yaru Jia, Yueming Wu, Deqing Zou, and Hai Jin. 2023. Code2Img: Tree-based Image Transformation for Scalable Code Clone Detection. *IEEE Transactions on Software Engineering* (2023).
[25] 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 (ASE'22)*. 1–12.
[26] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2021. FCCA: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability* 70, 1 (2021), 304–318.
[27] Yu-Liang Hung and Shingo Takada. 2020. CPPCD: A token-based approach to detecting potential clones. In *Proceedings of the 14th IEEE International Workshop on Software Clones (IWSC'20)*. 26–32.
[28] 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.
[29] Young-Bin Jo, Jihyun Lee, and Cheol-Jung Yoo. 2021. Two-Pass technique for clone detection and type classification using tree-based convolution neural network. *Applied Sciences* 11, 14 (2021), 1–18.
[30] 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.
[31] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundation of Software Engineering/ European Software Engineering Conference (ESEC/FSE'05)*. 187–196.
[32] 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.
[33] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*. 301–309.
[34] Thierry Lavoie, Michael Eilers-Smith, and Ettore Merlo. 2010. Challenging cloning related problems with GPU-based algorithms. In *Proceedings of the 4th International Workshop on Software Clones (IWSC'10)*. 25–32.
[35] Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee. 2016. CLORIFI: Software vulnerability discovery using code clone verification. *Concurrency and Computation: Practice and Experience* 28, 6 (2016), 1900–1917.
[36] 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.
[37] Hongliang Liang and Lu Ai. 2021. AST-path based compare-aggregate network for code clone detection. In *Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN'21)*. 1–8.
[38] Frank J. Massey. 1951. The Kolmogorov-Smirnov test for goodness of fit. *American Statistical Association* 46, 253 (1951), 68–78.
[39] Jean Mayrand, Claude Leblanc, and Ettore Merlo. 1996. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance (ICSM'96)*. 244–253.
[40] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI'16)*. 1287–1293.
[41] 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 (ESEC/FSE'21)*. 830–841.
[42] Gonzalo Navarro. 2001. A guided tour to approximate string matching. *Computing Surveys* 33, 1 (2001), 31–88.
[43] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the 2013 International Multiconference of Engineers and Computer Scientists (IMECS'13)*, Vol. 1. 380–384.
[44] Jayadeep Pati, Babloo Kumar, Devesh Manjhi, and Kaushal K. Shukla. 2017. A comparison among ARIMA, BP-NN, and MOGA-NN for software clone evolution prediction. *IEEE Access* 5, 1 (2017), 11841–11851.
[45] Nam H. Pham, Tung T. Nguyen, Hoan A. Nguyen, and Tien N. Nguyen. 2010. Detection of recurring software vulnerabilities. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. 447–456.
[46] Ross Quinlan. 1986. Induction of decision trees. *Machine Learning* 1, 1 (1986), 81–106.
[47] Chanchal K. Roy and James Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.
[48] 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.
[49] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling code clone detection to big code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 1157–1168.
[50] Sandip Sarkar, Dipankar Das, Partha Pakray, and Alexander Gelbukh. 2016. JUNITMZ at SemEval-2016 task 1: Identifying semantic similarity using Levenshtein ratio. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval'16)*. 702–705.
[51] 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 (ESEC/FSE'23)*. 784–795.
[52] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad M. 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.
[53] Kai S. Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[54] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2005. Data mining cluster analysis: Basic concepts and algorithms. *Introduction to Data Mining* 487, 1 (2005), 487–568.

[55] Vikas Thada and Vivek Jaglan. 2013. Comparison of jaccard, dice, cosine similarity coefficient to find best fitness value for web retrieved documents using genetic algorithm. *Innovations in Engineering and Technology* 2, 4 (2013), 202–205.

[56] 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.

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

[58] Yaoshu Wang, Jianbin Qin, and Wei Wang. 2017. Efficient approximate entity matching using jaro-winkler distance. In *Proceedings of the 2017 International Conference on Web Information Systems Engineering (WISE'17)*. Springer, 231–239.

[59] Yuekun Wang, Yuhang Ye, Yueming Wu, Weiwei Zhang, Yinxing Xue, and Yang Liu. 2023. Comparison and Evaluation of Clone Detection Techniques with Different Code Representations. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*. IEEE, 332–344.

[60] 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.

[61] Peter H. Westfall and Stanley Young. 1993. *Resampling-based multiple testing: Examples and methods for p-value adjustment*.

[62] 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.

[63] Yueming Wu, Siyue Feng, Wenqi Suo, Deqing Zou, and Hai Jin. 2023. Goner: Building Tree-Based N-Gram-Like Model for Semantic Code Clone Detection. *IEEE Transactions on Reliability* (2023).

[64] 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 (ASE'22)*.

[65] 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)*. 1000–1012.

[66] Sihan Xu, Ya Gao, Lingling Fan, Zheli Liu, Yang Liu, and Hua Ji. 2023. Lidetector: License incompatibility detection for open source software. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–28.

[67] 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.

[68] 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 (ESEC/FSE'18)*. 141–151.

[69] Deqing Zou, Siyue Feng, Yueming Wu, Wenqi Suo, and Hai Jin. 2023. Tritor: Detecting Semantic Code Clones by Building Social Network-Based Triads Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23)*. 771–783.

[70] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: A PDG-based code clone detector with approximate graph matching. In *Proceedings of the 35th International Conference on Automated Software Engineering (ASE'20)*. 931–942.