Goner: Building Tree-based N-gram-like Model for Semantic Code Clone Detection

Yueming Wu, Siyue Feng, Wenqi Suo, Deqing Zou, Hai Jin, *Fellow, IEEE*

***Abstract*—Code clone detection refers to the detection of code fragments that are functionally similar. As software engineering progresses, the significance of code clone detection continues to grow. A number of code clone detection techniques have been designed. Among these methods, tree-based code clone detection approaches have the ability to discover semantic code clones. However, given the intricate nature of tree structures, they consume plenty of time to complete the tree analysis, thus cannot scale to large-scale code scanning. In this paper, we propose a novel tree-based scalable semantic code clone detection method by transforming the heavy-weight tree processing into efficient N-gram-like subtrees analysis. Specifically, we build a variant of N-gram model to partition the original complex tree into small subtrees. After collecting all subtrees, we divide them into different groups according to the positions of the subtree nodes, and then calculate the similarity of the same group between two functions one by one. Similarity scores of all groups are made up of a feature vector. Given feature vectors, we train a machine learning model for semantic code clone detection. We implement *Goner*, a scalable tree-based semantic code clone detection system. To showcase the effectiveness of *Goner*, we conducted evaluations on two extensively utilized datasets, namely BigCloneBench and Google Code Jam. The experimental results unequivocally indicate that *Goner* outperforms our comparative systems (*i.e., SourcererCC*, *RtvNN*, *Deckard*, *ASTNN*, *TBCNN*, *CDLH*, *Amain*, *FCCA*, *DeepSim*, and *SCDetector*). Additionally, in the context of scalability, *Goner* demonstrates remarkable speed, being approximately 56 times faster than another advanced tree-based tool, namely *ASTNN*, when it comes to identifying semantic code clones.**

***Index Terms*—Semantic Code Clones, Abstract Syntax Tree, N-gram**

I. Introduction

CODE clone is the phenomenon of copying the entire source code or code fragments. In reality, code clones are divided into syntactic clones and semantic clones. Syntactic clones are usually found when copying and pasting code and are classified into three types in descending order of similarity, namely Type-1 (textual similarity), Type-2 (lexical similarity), and Type-3 (syntactic similarity). Semantic clones are usually introduced when different code syntaxes are used to achieve the same functionality, which is Type-4 (semantic similarity). As the computer and software fields continue to advance and evolve over time, developers often need to write and handle large amounts of code. Code clone can often save developers' time and effort to a great extent, making code cloning common. However, the drawbacks brought by code clone cannot be ignored, which is the consequent increase in maintenance costs. For example, if the copied code contains any vulnerabilities, code cloning may amplify the propagation of these vulnerabilities. Therefore, code clone detection becomes increasingly important.

Recently, there have been designed many code clone detection techniques where token-based methods are the most scalable. For example, *CCFinder* [1] transforms the tokens extracted from the source code by means of certain pre-defined transformation rules. However, it can only detect Type-1 and Type-2 code clone due to their simple code transformation rules. To handle Type-3 code clone, *SourcererCC* [2] computes the code similarity by analyzing the overlap between tokens. While it is capable of identifying certain Type-3 code clones, it lacks the capability to detect Type-4 code clone (*i.e.,* semantic code clone). In fact, compared to the former three types of code clone, Type-4 code clones are more difficult to be distinguished since they are syntactically dissimilar. To address the detection of Type-4 code clones, researchers propose to distill the program semantics into different representations (*e.g.,* abstract syntax tree and control flow graph) and use them to calculate the semantic similarity of different code fragments. Since the consideration of program semantics, graph-based detection methods [3]–[7] and tree-based methods [8]–[12] can both deal with semantic code clones. However, typical graph analysis is heavy-weight which means that they may require lots of time to finish code clone detection. For tree-based methods, a tree generated by a function of a few lines of code may contain dozens of nodes. For example, Figure 3 provides a visual representation of the abstract syntax tree associated with the GCD method (Type-4 method) depicted in Figure 1. The simple four lines of code can generate an abstract syntax tree with 36 nodes. In other words, the tree structures are also complex, making it difficult to complete large-scale code clone detection. Therefore, there is a growing need for a code clone detector that possesses both semantic clone detection capabilities and high scalability.

The focus of this paper is to propose a novel and scalable tree-based code clone detector that can detect semantic code clones at a large scale. We specifically tackle two significant challenges.

Yueming Wu, Siyue Feng, Wenqi Suo, and Deqing Zou (corresponding author) are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China (e-mail: wuyueming21@gmail.com, fengsiyue@hust.edu.cn, suowenqi@hust.edu.cn, deqingzou@hust.edu.cn).

Hai Jin is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China (e-mail: hjin@hust.edu.cn).

- *Challenge 1: How to simplify the intricate structure of a tree while preserving its rich semantic information?*
- *Challenge 2: How to develop a code clone detection process that is streamlined and efficient in handling semantic clones at scale?*

To overcome the first challenge, we propose a variant of N-gram model to split the original abstract syntax tree into small subtrees to simplify the tree complexity. The traditional N-gram method is mainly used in natural language to calculate the similarity between two texts. In our method, we start from the root node in a tree, and the *n* nodes connected together constitute a subtree. Assuming *n* is 2, then the parent node and each of its child nodes form a subtree. In this way, we partition the complex abstract syntax tree into many n-node subtrees. By performing statistical comparison operations on these subtrees, we can implement the detection of semantic clones in a scalable manner while preserving the semantic information of the program.

To address the second challenge, we design a novel similarity measurement technique to construct feature vectors and use them to train a machine learning model for efficient code clone detection. Specifically, given all collected subtrees, we first classify them into different groups based on the positions of the subtree nodes. The similarity of each group between two methods corresponds to a feature, and the similarity scores of all groups constitute the feature vector. In other words, given two methods, we can output a feature vector after computing the similarity scores of all groups. Finally, these feature vectors are put into a machine learning model for training to obtain a clone detector for fast and accurate semantic code clone detection.

We have developed a prototype system namely *Goner* and extensively evaluate the system using two prominent datasets: *BigCloneBench* (BCB) [13], [14] and *Google Code Jam* (GCJ) [15]. The experimental results demonstrate that *Goner* greatly improves the accuracy of clone detection compared to ten state-of-the-art code clone detectors. These detectors include two token-based methods (*SourcererCC* [2] and *RtvNN* [16]), five tree-based methods (*Deckard* [8], *ASTNN* [10], *TBCNN* [17], *Amain* [18], and *CDLH* [9]), and three graph-based methods (*SCDetector* [19], *DeepSim* [6], and *FCCA* [20]). In particular, it performs well in detecting Type-4 clones. Moreover, compared to an advanced tree-based code clone detector (*i.e., ASTNN*), *Goner* is approximately 51 times faster during the training phase and around 120 times faster during the predicting phase.

In summary, this paper provides the following contributions:

- We propose a novel N-gram-like model to transform the complex tree analysis into simple and efficient subtree analysis.
- We implement a prototype system, *Goner*[1], a tree-based semantic code clone detector. Building N-gram-like model makes *Goner* suitable for scanning large-scale code clones.
- We check the ability of *Goner* by conducting comparative experiments on BigCloneBench dataset and Google Code Jam dataset. Experimental results validate that *Goner* performs better than ten state-of-the-art systems.

[1]https://github.com/SiyueFeng99/GonerCode.

Paper organization. The remainder of the paper is organized as follows. Section II presents the background and motivation. Section III shows our system. Section IV reports the experimental results. Section V discusses the future work. Section VI describes the related work. Section VII concludes the present paper.

II. Background and Motivation

Before introducing our proposed system, it is necessary to establish a clear understanding of certain definitions that will be utilized throughout the paper, including definitions of clone types and code granularity.

A. Clone Type

Code clone is the phenomenon of copying the entire source code or code fragments. According to the level of similarity, clone clone is generally divided into the following four types [21], [22]:

- **Type-1 (textual similarity)**: Code fragments that are identical, except for variations in white-space, layout, and comments.
- **Type-2 (lexical similarity)**: Code fragments that are identical, except for variations in identifier names and lexical values, in addition to the differences in Type-1 clones.
- **Type-3 (syntactic similarity)**: Code snippets that exhibit syntactic similarities but at the statement level. Alongside Type-1 and Type-2 clone differences, these fragments may have added, modified, or removed statements relative to one another.
- **Type-4 (semantically similarity)**: Code fragments that are syntactically dissimilar but implement the same functionality.

To visualize the different types of clones, Figure 1 gives a showcasing example of clones from Type-1 to Type-4. These clones illustrate different variations or instances related to the original method, which calculates the greatest common divisor of two numbers. The Type-1 code clone is exactly the same as the original method, nothing has changed. Type-2 code clone has the same code fragments, but with different identifier names (*i.e.,* m and n instead of a and b). Type-3 code clone has the same syntax, but the order of statements, the method name, and some variable types are different. Type-4 code clone replaces the while loop with a function call, implementing the same functionality with a completely different syntax structure. This type of code clone is also called semantic code clone and is the most difficult to discover since the code structure may change a lot.

B. Code Granularity

We also give a definition of the granularity of the code, which refers to the size of code fragments in the whole study. Depending on the size of the granularity unit, it can be divided into Token, Line, Function, File, and Program. A line consists

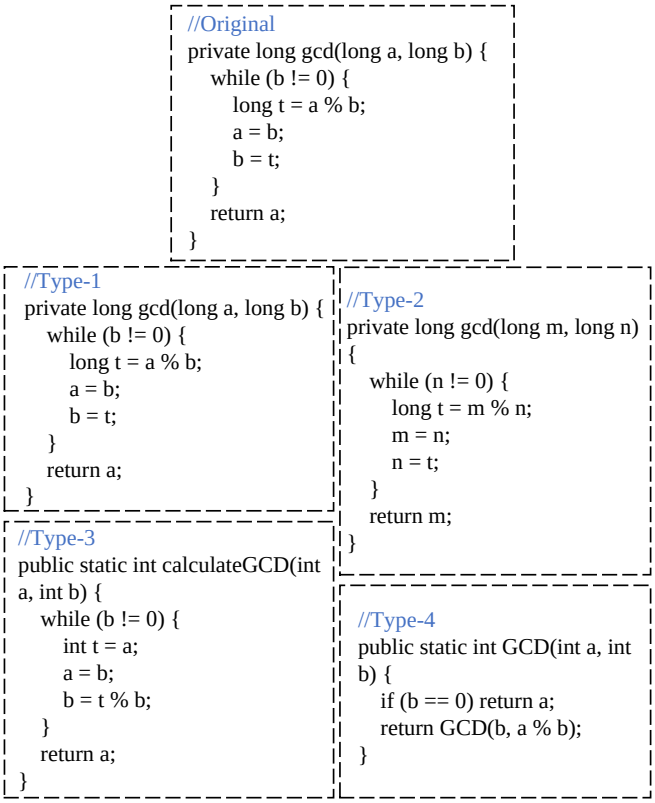


Fig. 1. Examples of different clone types

of multiple tokens, and many lines in turn make up functions. A program is a set of files encompassing functions. From the largest granularity program-level, to the smallest granularity token, code cloning can occur on any granularity unit. Since cloning of entire files or programs rarely occurs, file-level and program-level are too coarse on code clone detection. While the clone pairs detected at line-level and token-level may not hold significant meaning (*e.g.,* simple statements such as '*int i = 0;*' and '*int j = 0;*' could be identified as a clone pair). While the clone at function-level can guarantee the cloning of complete functions, so we use function-level as the processing granularity.

C. Motivation

To illustrate more clearly how our approach is proposed, we present a simple example in this part. Figure 1 shows that both Original and Type-4 methods implement the functionality of calculating the GCD of two numbers by different syntactic structures, which is a typical semantic code clone.

The traditional N-gram method is mainly used in natural language to calculate the similarity between two texts. N-gram slices the strings by length n, and all substrings of length n in the original sentence are obtained. The distance between two texts is calculated by counting the number of identical substrings. The closer the distance, the more similar the two sentences are. For two sentences S and T, N-gram defines the distance of two sentences as follows:

$$Distance = |GN(S)| + |GN(T)| - 2 \times |GN(S) \cap GN(T)| \quad (1)$$

where $GN(S)$ (or $GN(T)$) denotes the set of substrings obtained by N-gram segmentation of length N for sentence S (or T). $|GN(S)|$ and $|GN(T)|$ denote the number of substrings in the two sentences, respectively. $|GN(S) \cap GN(T)|$ denotes the number of common substrings in the two sentences.

while (	(b	b !=	!= 0	0)

Fig. 2. substrings obtained by the '*while (b != 0)*' statement

We first illustrate how to calculate the distance between these two methods using traditional N-gram approach. If we set n as 2, the statement '*while (b != 0)*' will be partitioned into five substrings as shown in Figure 2. After collecting all substrings from the Original method and Type-4 method shown in Figure 1, we find that the Original method has a total of 37 substrings, and Type-4 method has a total of 31 substrings. In addition, they have 10 substrings in common. In other words, $|GN(S)| = 37$, $|GN(T)| = 31$, and $|GN(S) \cap GN(T)| = 10$. The distance between the two methods calculated using the traditional N-gram method is 37+31-2×10 = 48.

In order to consider the program information of methods, we first extract the corresponding abstract syntax trees and then use a 2-gram-like approach to compute the distance. Figure 3 shows the abstract syntax tree corresponding to the Type-4 method in Figure 1. Specifically, we partition the abstract syntax tree into some subtrees with two nodes. In fact, in this case, what we analyze is the edges of the tree. After statistical analysis, we observe that original method has 49 two-node subtrees (*i.e.,* edges), and Type-4 method has 35 two-node subtrees. In total, there are 27 common subtrees. In other words, $|GN(S)| = 49$, $|GN(T)| = 35$, and $|GN(S) \cap GN(T)| = 27$. The resulting similarity distance is 49+35-2×27 = 30, which is smaller than 48 calculated by using traditional N-gram method on source code.

In one word, the similarity distance calculated using tree-based 2-gram-like model is smaller than that obtained using traditional code-based 2-gram model. With a certain threshold value, tree-based 2-gram-like model may detect the code pair of Original and Type-4 methods in Figure 1 as a code clone. In other words, using the principle of N-gram to segment the tree into subtrees with n nodes may hold the potential to identify semantic code clones. Building upon this observation, we develop a tree-based N-gram-like model for semantic code clone detection.

III. System

In this section, we present our code clone detection system, which we refer to as *Goner*.

A. Overview

As described in Figure 4, *Goner* consists of four main phases: *AST Generation*, *AST Division*, *Feature Extraction*, and *Classification*.

- ***AST Generation***: The purpose of this phase is to statically analyze the input program and obtain an *abstract syntax*

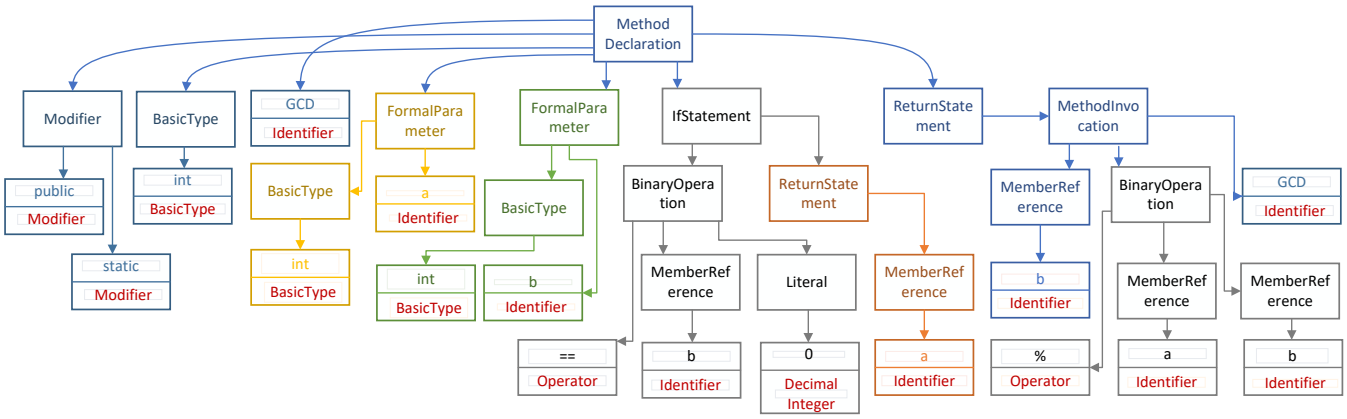


Fig. 3. Abstract syntax tree of the Type-4 method in Figure 1

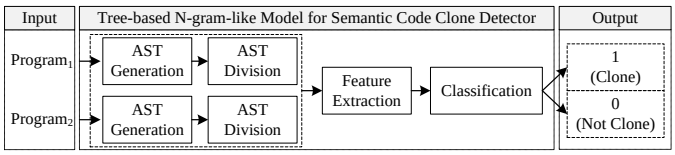


Fig. 4. System architecture of *Goner*

tree (AST) for each code. The input to this phase is a method while the output is an AST.

- ***AST Division***: The purpose of this phase is to build a N-gram-like model to partition the original AST into subtrees. The input of this phase is an AST while the output is the number of various subtrees.
- ***Feature Extraction***: The purpose of this phase is to construct a feature vector by calculating the similarity of different types of subtrees. The input of this phase is subtrees of two methods while the output is a feature vector.
- ***Classification***: The purpose of this phase is to train a code clone detector by using some feature vectors and then use the detector to find code clones. The input of this phase is a feature vector of two methods while the output is the corresponding label (*i.e.,* clone or not clone).

B. AST Generation

In this paper, our goal is to apply the N-gram principle to partition the AST of each method into subtrees. Therefore, we need to parse the method to obtain the corresponding AST. Given that our experimental dataset (*i.e.,* BigCloneBench [13], [14] and Google Code Jam [15]) comprises *Java* code, we leverage *Javalang* [23] for conducting static analysis.

Fig. 5. The string divided by 2-gram

In order to give all nodes a distinct type, after collecting the AST of all methods in BigCloneBench and Google Code Jam datasets, we perform a deep analysis to count the types of all nodes in the AST. After analysis, for nodes except leaf nodes, we identify a total of 57 distinct types. We use the type names to represent these nodes. For leaf nodes, we first tokenize all methods in BigCloneBench and Google Code Jam datasets and then count the types of all tokens. From the results, we find that the vast majority of tokens fall into the 14 types, including "*Annotation*", "*BasicType*", "*BinaryInteger*", "*Boolean*", "*DecimalFloatingPoint*", "*DecimalInteger*", "*HexFloatingPoint*", "*HexInteger*", "*Identifier*", "*Keyword*", "*Modifier*", "*OctalInteger*", "*Operator*", and "*Separator*". So we use these 14 types as token types. For the rare tokens that do not fall into one of these 14 types, we classify them as "*Null*" type. In this way, we get a total of 15 token types. The AST presented in Figure 3 is derived from parsing the GCD method depicted in Figure 1. The leaf nodes are represented by the types corresponding to the token values in the leaf nodes, shown in red in figure 3.

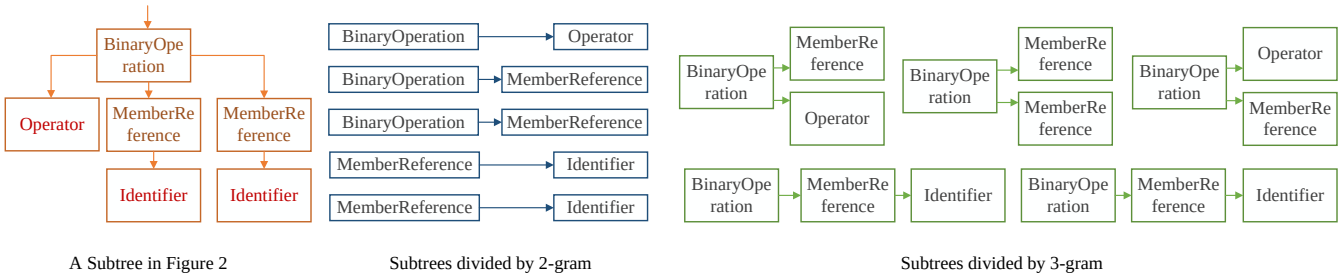
Algorithm 1 AST Division

Input: an AST and the value of n in the n-gram.
Output: $SubtreeDict$, the mapping of each subtree to its number.

```
SubtreeList ← []
if n == 2 then
    GET2GRAM( ASTRootNode, SubtreeList)
end if
if n == 3 then
    path ← []
    GET3GRAM_1(ASTRootNode, SubtreeList, path)
    GET3GRAM_2(ASTRootNode, SubtreeList)
end if
for each subtree in SubtreeList do
    if subtree in SubtreeDict then
        SubtreeDict[subtree] ← SubtreeDict[subtree] + 1
    else
        SubtreeDict[subtree] ← 1
    end if
end for
```

C. AST Division

The principle of N-gram segmentation is relatively simple. For a string of length l, it is to divide the ith word and the $i+1$, $i+2$ ... $i+n-1$ words into a substring. Starting from



A Subtree in Figure 2 Subtrees divided by 2-gram Subtrees divided by 3-gram

Fig. 6. Subtrees divided by 2-gram and 3-gram

the first word, it has been divided until the $l-n$th word, so a total of $l-n$ substrings can be obtained. For example, Figure 5 shows a string that needs to be divided, and each framed place is the substring obtained by the 2-gram division.

After a series of processing in the AST generation phase, we obtain the AST represented by node types. Similarly, the segmentation principle of N-gram is applied to AST. Starting from the root node, the connected n nodes are the subtrees obtained by division. For example, the first part of Figure 6 represents a subtree of Figure 3. If we perform a 2-gram partition on this subtree, the subtrees obtained is shown in the second part of Figure 6, and if we perform a 3-gram partition on the subtree, the subtrees obtained is shown in the third part of Figure 6. Then we count the number of each subtree and record it. For example, there are three different subtrees in the second part of Figure 6: one for *BinaryOperation-Operator*, two for *BinaryOperation-MemberReference*, and two for *MemberReference-Identifier*.

Algorithm 2 Get2gram

Input: a $node$ in AST, $SubtreeList$ which contains subtrees.

function GET2GRAM($node, SubtreeList$)
  for each $childnode$ in $node.children$ **do**
    Add the $(node, childnode)$ into the $SubtreeList$
    GET2GRAM($childnode, SubtreeList$)
  end for
end function

Algorithm 3 Get3gram_1

Input: a $node$ in AST, $SubtreeList$ which contains subtrees, and $path$ from the root node to the leaf node.

function GET3GRAM_1($node, SubtreeList, path$)
  Add the $node$ into the $path$
  if $path.lenth >= 3$ **then**
    Add the $(path[-3], path[-2], path[-1])$ into the $SubtreeList$
  end if
  for each $childnode$ in $node.children$ **do**
    GET3GRAM_1($childnode, SubtreeList, path$)
  end for
  Pop $path[-1]$ from the $path$
end function

We describe the AST segmentation algorithm in Algorithm 1. The input to the algorithm is an AST and the value of n in the n-gram. The output is subtrees after splitting using N-gram and the mapping of each subtree to its number, which we call SubtreeDict. The Get2gram method, the Get3gram_1 method, and the Get3gram_2 method are described in Algorithms 2-4. The functionality of Get2gram method is to extract the subtrees obtained from the 2-gram division of the AST. The functionality of Get3gram_1 method is to extract the subtrees of the relation *parent-child-grandchild* (such as the two in the bottom row of the third part in Figure 6) obtained from the 3-gram division. The functionality of Get3gram_2 method is to extract the subtrees of the relation *parent-child-child* (such as the three in the top row of the third part in Figure 6) obtained from the 3-gram division.

Algorithm 4 Get3gram_2

Input: a $node$ in AST, $SubtreeList$ which contains subtrees.

function GET3GRAM_2($node, SubtreeList$)
  for two different child nodes of $node$: $childnode1, childnode2$ **do**
    Add the $(node, childnode1, childnode2)$ into the $SubtreeList$
  end for
  for each $childnode$ in $node.children$ **do**
    GET3GRAM_2($childnode, SubtreeList$)
  end for
end function

D. Feature Extraction

After processing in the AST division stage, an AST is split into different numbers of subtrees with n nodes. We then classify each subtree into different groups based on the position of subtree nodes. A subtree with n nodes can belong to n groups according to their positions.

When n is two, a node type can be in the first position or in the second position. So in addition to the types of leaf nodes, each type can be divided into two groups according to the position. Since the leaf nodes are only in the second position, the type of each leaf node is classified into one group. As shown in Figure 7, *BinaryOperation* can be divided into two groups: *BinaryOperation* at the first position and *BinaryOperation* at the second position. For example, since *BinaryOperation* is at the second position of the subtree *IfStatement-BinaryOperation*, this subtree is classified into the group of *BinaryOperation* at the second position. When *BinaryOperation* is in the first position, there are three kinds of subtrees that belong to this group, *BinaryOperation-Operator*, *BinaryOperation-MemberReference*, *BinaryOperation-Literal*. When *BinaryOperation* is in the second position, there are four kinds of subtrees that belong to this group. The number of different subtrees in each group is counted to form

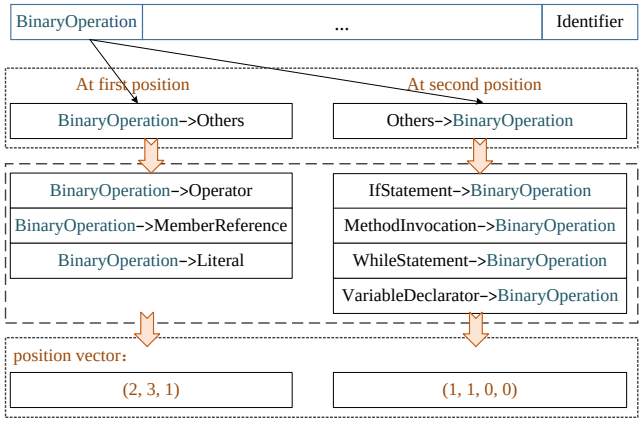


Fig. 7. The position vectors generated by 2-gram

the position vector of the group. That is, each number in the position vector is actually the number of times each subtree in this group appears in the AST. For example, the position vector (2, 3, 1) in the Figure 7 indicates that the *subtree BinaryOperation-Operator* appears two times, *BinaryOperation-MemberReference* appears three times, and *BinaryOperation-Literal* appears one time.

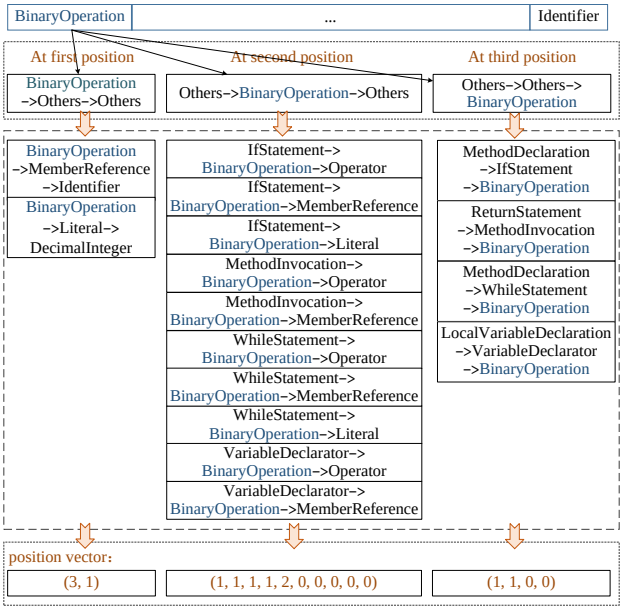


Fig. 8. The position vectors of parent-child-grandchild relationship generated by 3-gram

However, for the subtree obtained by 3-gram partitioning, the grouping is a bit different from that of 2-gram. Since the three nodes can be a *parent-child-grandchild* relationship, such as the two in the bottom row of the third part in Figure 6. It can also be a *parent-child-child* relationship, such as the three in the top row of the third part in Figure 6. In that case, the relationship of the three nodes needs to be considered first when we group them.

If the relationship of three nodes is *parent-child-grandchild*, then similar to the 2-gram, the subtree is grouped according to the position of each node. A node type will then be divided into three groups. As shown in Figure 8, there are two kinds of subtrees that belong to the group that *BinaryOperation* at the first position, ten kinds of subtrees belong to the group that *BinaryOperation* at the second position, and four kinds of subtrees belong to the group that *BinaryOperation* at the third position.

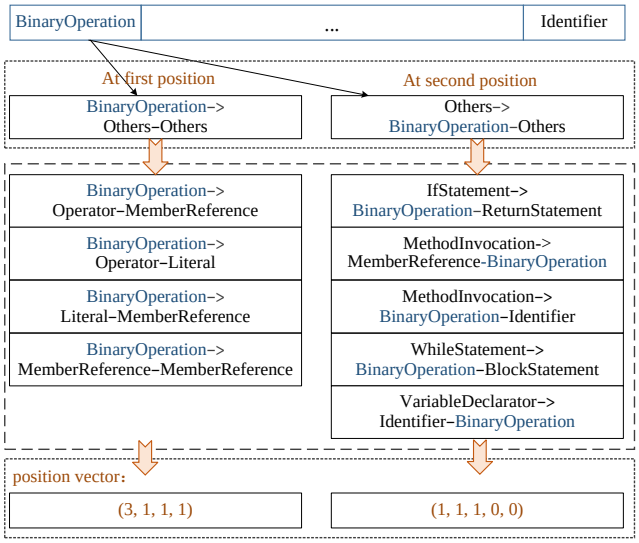


Fig. 9. The position vectors of parent-child-child relationship generated by 3-gram

If the relationship of three nodes is *parent-child-child*, there is no difference between the two child nodes in the positional relationship, they both belong to the second position. In this way, the first subtree and third subtree in the top row of the third part in Figure 6 are actually the same kind of subtree. Then the node types in this relationship are divided into two groups according to position, which are the first position and the second position. As shown in Figure 9, the subtree *MethodInvocation-MemberReference-BinaryOperation* contains *BinaryOperation* in the second position, so this subtree is grouped into the *BinaryOperation* in the second position. As we can see in the Figure 9, the group *BinaryOperation in the first position* has four kinds of subtrees, and the group *BinaryOperation* in the second position has five kinds of subtrees.

According to such a principle, we aggregate the subtrees obtained from all AST splits and group these subtrees. Each subtree has a fixed position, and each group has a fixed number of subtrees. The algorithm for subtree grouping is described in Algorithm 5. The input of Algorithm 5 is the subtrees obtained by AST division of all methods, and the output is the order of all subtrees after grouping, which we call SubtreeOrderList, and the number of subtrees contained in each group, which we call GroupLenthList. We create a reverse index for the SubtreeOrderList and let each subtree point to its index to get the PositionDict.

All subtrees obtained by partitioning the AST of a method are classified and the number of various subtrees contained

in each group is counted to form a vector, which is the position vector of this group. That is, each number in the position vector is actually the number of times each subtree in this group appears in the AST. For example, in Figure 7, Figure 8, and Figure 9, the vector below each group is the position vector of the corresponding group. Different *n* means different number of groups, and subsequently different number of position vectors. For example, when *n* is 2, there are 116 position vectors in total. When *n* is 3, there are 268 position vectors in total.

Algorithm 5 Get PositionDict and GroupLenthList

Input: $AllSubtrees$ obtained by AST division of all methods.
Output: $SubtreeOrderList$, the order of all subtrees after grouping and $GroupLenthList$, the number of subtrees contained in each group.

for each $subtree$ in $AllSubtrees$ **do**
  if the relationship of nodes in $subtree$ is *parent-child-grandchild* **then**
    $parent \leftarrow subtree[0]$
    $child \leftarrow subtree[1]$
    $grandchild \leftarrow subtree[2]$
    Add the $subtree$ into the $group0[parent]$
    Add the $subtree$ into the $group1[child]$
    Add the $subtree$ into the $group2[grandchild]$
  else if the relationship of nodes in $subtree$ is *parent-child-child* **then**
    $parent \leftarrow subtree[0]$
    $child1 \leftarrow subtree[1]$
    $child2 \leftarrow subtree[2]$
    Add the $subtree$ into the $group3[parent]$
    Add the $subtree$ into the $group4[child1]$
    if $child1! = child2$ **then**
      Add the $subtree$ into the $group4[child2]$
  end if
end for
for each $group$ in $(group0, group1, group2, group3, group4)$ **do**
  for each $node$ in $group$ **do**
    for each $subtree$ in $group[node]$ **do**
      Add the $subtree$ into the $SubtreeOrderList$
    end for
    Add the $group[node].lenth$ into the $GroupLenthList$
  end for
end for

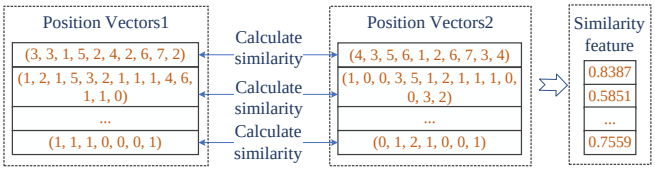


Fig. 10. Calculate the similarity of two methods to obtain the feature vector

According to this operation, after obtaining the position vectors of all groups of each method, the similarity calculation is performed on the vectors of the same group corresponding to two methods. The similarity results of all groups compose the similarity vector of the two methods. Figure 10 shows the similarity calculation for the position vectors of the same group of two methods. Take the first vector for similarity calculation and get a similarity result. Then take subsequent vectors until all vectors are taken. When *n* is 2, there are 116 groups in total, so the number of similarity calculation results obtained is 116. These 116 results form a 116-dimensional similarity vector, which is the feature vector of the two methods. When *n* is 3, the feature vector of the two methods is 268-dimensional. As for the similarity calculation method, we evaluate the similarity between two vectors by using cosine similarity, which calculates the cosine of the angle between the two vectors. The calculation formula is as follows:

$$cos(\theta) = \frac{\Sigma_{i=1}^{n} A_i B_i}{\sqrt{\Sigma_{i=1}^{n} A_i^2} \sqrt{\Sigma_{i=1}^{n} B_i^2}} \quad (2)$$

E. Classification

In the classification phase, we choose to use machine learning models for the classification of code pairs because of their strong classification capability. Machine learning has a wide range of applications in many fields such as data analysis and mining, pattern recognition, bioinformatics, and so on. Specifically, we select four common machine learning algorithms (*i.e., k-nearest neighbor* (KNN) [24], *random forest* (RF) [25], *decision tree* (DT) [26], *adaptive boosting* (Adaboost) [27], *gradient boosting decision tree* (GDBT) [28], and *extreme gradient boosting* (XGBoost) [29]) to examine the detection effectiveness of *Goner*. These algorithms are the more popular machine learning algorithms that are often used for classification problems. After obtaining the similarity vectors on all code pairs including clone pairs and non-clone pairs, we treat these vectors with labels as training data for a machine learning model employed as a code clone detector. Given a new similarity vector obtained from a pair of codes to be detected, the detector can output the corresponding label (*i.e.,* clone or not clone).

IV. Experiment

In this section, we strive to address the following research questions:

- *RQ1: how Goner performs under various algorithm choices and different n of N-gram?*
- *RQ2: How effective is Goner in detecting different types of code clones compared to other code clone detectors?*
- *RQ3: Does the consideration for location information in the feature extraction phase have a positive effect on detection?*
- *RQ4: Does the use of machine learning algorithms have a positive effect on detection?*
- *RQ5: What is the runtime overhead during the process of code clone detection by Goner?*
- *RQ6: Why is Goner effective in detecting semantic code clones?*

A. Experimental Settings

1) Dataset

To answer the six questions posed above, we run our experiments on two datasets. The first dataset is a widely used dataset (*i.e.,* BigCloneBench [13]). It contains more than eight million labeled clone pairs with code granularity at the function level, which can well meet the needs of our experiments. Since it is difficult to distinguish the boundary between

Type-3 and Type-4, after textual and lexical normalization of these two clone types, the similarity at line-level and token-level is calculated. Type-3 and Type-4 are further divided into three sub-categories according to the size of similarity. These sub-categories include: i) *Strongly Type-3* (ST3), displaying a similarity range of 70-100%, ii) *Moderately Type-3* (MT3), displaying a similarity range of 50-70%, and iii) *Weakly Type-3/Type-4* (WT3/T4), displaying a similarity range of 0-50%.

The second dataset is Google Code Jam [15], an online programming competition from Google that contains 1,669 projects pertaining to 12 distinct competition problems. The different competition projects that solve different problems are semantically and syntactically dissimilar to each other, so we regard them as non-clone pairs. Different projects for the same competition problem are written by different programmers, so they are syntactically different and semantically similar, and we treat them as semantic clone pairs.

For the BigCloneBench dataset, we randomly select 270,000 clone pairs from the extensive pool of eight million clone pairs to ensure that an equal number of non-clone code pairs, amounting to 270,000, are included in our analysis. The clone pairs we selected include 48,116 clone pairs of *Type-1* (T1), 4,234 clone pairs of *Type-2* (T2), 21,395 clone pairs of *Strongly Type-3* (ST3), 86,341 clone pairs of *Moderately Type-3* (MT3), and 109,914 clone pairs of *Weakly Type-3/Type-4* (WT3/T4). For the Google Code Jam dataset, we selected all 275,570 semantic clone pairs and randomly selected 270,000 non-clone pairs from a pool of 1,116,376 non-clone pairs.

To conduct training and testing, we employ the ten-fold cross-validation method. It means that we divide the entire dataset into ten equal parts, and each part is used as the testing set while the remaining parts are used for training the model. We record F1, precision, and recall for each validation, and take the average of ten records as the final clone detection metrics.

2) Implementations

During the AST generation phase, we use the Python library *Javalang* [23] to obtain the abstract syntax tree of the methods since the experimental dataset consists of code written in the Java programming language. The cosine similarity is calculated using the Python library *Sklearn* [30] in the feature extraction stage. In the classification stage, *Sklearn* is also used to implement KNN, Random Forest, and Decision Tree classification algorithms. All the experiments were carried out on a server running the Ubuntu 20.04.2 LTS operating system, with 64GB RAM and an Intel (R) Xeon (R) Gold 6248R CPU with 8 cores.

3) Comparative Systems

There are already various methods for detecting code clones, but most of them are not publicly available. Therefore, we select the following state-of-the-art and open source code clone detection methods to measure the advantages and disadvantages of *Goner* by comparing the detection results and the running overhead.

SourcererCC [2]: a popular token-based code clone detector with the capability to handle large-scale code. *Deckard* [8]: a popular AST-based code clone detector which clusters the vectors of AST subtree. *RtvNN* [16]: a popular RNN-based code clone detector which encodes source code tokens and ASTs. *ASTNN* [10]: a popular AST-based code clone detector which splits a large tree into certain statement trees and trains an RNN model to detect code clones. *TBCNN* [17] : a popular clone detection detector based on AST and utilizes convolutional neural network. *CDLH* [9] : a popular clone detection method based on AST and utilizes long short-term memory network. *Amain* [18] : a popular clone detection detector based on AST and builds Markov Chains Model. *SCDetector* [19] : a popular graph-based code clone detector which extracts the CFG of a method and apply centrality analysis to detect code clones. *DeepSim* [6] : an advanced clone detection tool that employs a deep neural network and based on graph. *FCCA* [20] : an advanced clone detection tool based on graph that uses hybrid code representations with high accuracy.

For the parameter settings[2] of these tools, we select the parameters reported in their published papers since they can perform best with these parameters.

4) Metrics

As in previous work [6], [19], we adopt the widely used metrics, *Precision* (P), *Recall* (R), and *F-measure* (F1), to measure the detection effectiveness of *Goner*.

B. RQ1: Contrast of Diverse Methods

To illustrate the effectiveness of different methods and different parameters in detecting clones, we set up comparison experiments in this subsection. We select 270,000 clone pairs and an equal number of 270,000 non-clone pairs from BCB as we mentioned in the Experimental Settings to complete the experiment. In the subtree segmentation stage, n is taken as two and three, respectively. When n is taken as two, 116-dimensional similarity features will be obtained in the feature extraction stage. When n is three, 268-dimensional similarity features will be obtained. The similarity features are put into various machine learning algorithms, including KNN, RF, DT, Adaboost, GDBT, and XGBoost. They are leveraged to train and save the models. As for the selection of parameters, we choose one and three as the neighbor parameters of KNN because these two numbers are the most commonly used. Similar to [18], [31], [32], we use the default parameters for RF, Adaboost, GDBT, and XGBoost.

From Figure 11, it can be seen that both F1 score, precision and recall are the best results obtained by performing 2-gram segmentation on AST using XGBoost to train similarity features. For example, when 2-gram is selected for subtree segmentation, the F1 scores of Random Forest algorithm, Decision Tree algorithm, 1NN algorithm, 3NN algorithm, Adaboost algorithm, GDBT algorithm, and XGBoost algorithm are 98.15%, 96.34%, 97.39%, 96.87%, 97.67%, 96.53%, and 98.80%, respectively. when 3-gram is selected for subtree segmentation, the F1 scores of these four machine learning algorithms are 98.11%, 96.39%, 97.34%, and 96.78%, 97.62%, 96.49%, 98.77%, respectively. The random forest algorithm is actually a combination of multiple decision trees. XGBoost

[2]The parameters are on website: https://github.com/SiyueFeng99/GonerCode.

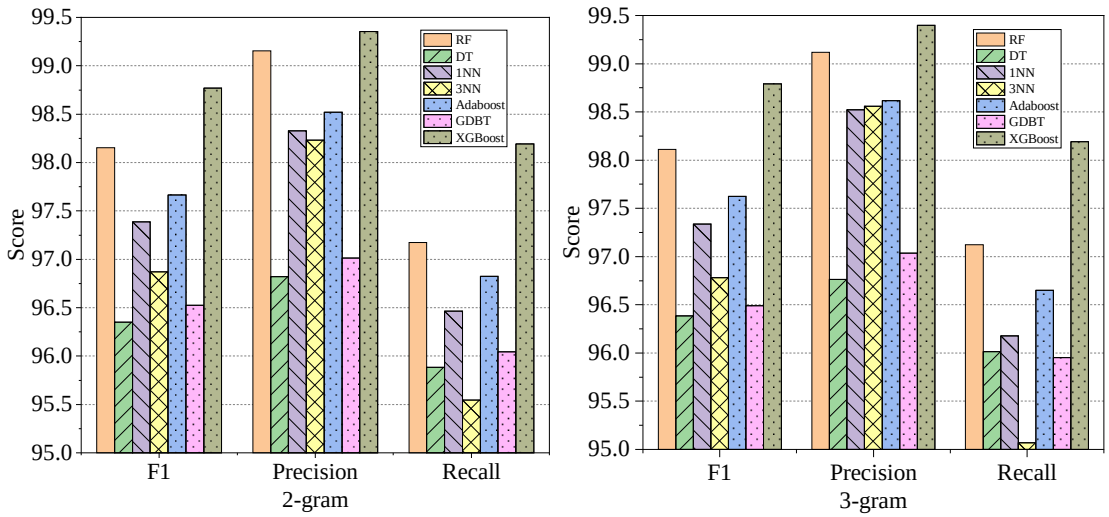


Fig. 11. F1 score, Precision and Recall of clone detection using different machine learning classification methods and different parameters

is an integrated learning method that implements classification and regression tasks by building multiple decision trees. Therefore, the prediction results of XGBoost algorithm are relatively outstanding. However, the running time of XGBoost is relatively high compared to RF, which also has ideal detection results. For example, XGBoost takes 113.52 seconds to complete the training phase, while RF takes only 87.98 seconds to complete. Under the premise of ensuring similar detection results, using RF can be more scalable. Therefore, we choose to use RF to complete the subsequent comparison experiments.

The difference between the 2-gram and 3-gram effects is not significant. In fact, when we adopt 2-gram to extract the subtrees, we can obtain 116 position vectors and the largest dimension is 34. For 3-gram, the number of position vectors we obtained is 268 and the largest dimension is 337. In the AST division phase, 2-gram takes 12 seconds to complete the operation on 73,319 files, while 3-gram takes 76 seconds to complete it, almost triple the time. In the feature extraction phase, 2-gram takes only 2.5 seconds to extract the similarity vectors of one million code pairs, while 3-gram takes 4.45 seconds to complete. In other words, 2-gram model consumes less runtime and requires less memory. Therefore, we use 2-gram to commence our comparative experiment.

The answer to RQ1: *Goner* can achieve ideal detection effectiveness by using 2-gram in the subtree segmentation stage and RF algorithm in the classification stage.

C. RQ2: Overall Effectiveness

In this subsection, we discuss the overall performance of *Goner* in detecting code clones compared to ten state-of-the-art code clone detectors. From the experimental results in the previous subsection, it can be seen that the detection effect of using 2-gram in the subtree segmentation stage and using the RF algorithm in the classification stage is ideal, so in this part we use 2-gram and Random Forest algorithm to complete the comparison.

1) Results on GCJ

First, we evaluate the effectiveness on the GCJ dataset. As we mentioned in the Experimental Settings, all clone pairs in the GCJ dataset are semantic clones. The comparative detection results of ten comparative systems and *Goner* are shown in Table I.

As can be seen from the results in the Table I, *Goner* achieves a recall, precision, and F1 score of 92%. Compared to other clone detection tools, *Goner*'s recall and F1 scores are highest, the precision is higher than most comparison tools.

For **Token-based** approaches, *SourcererCC* has high precision but very poor recall. This is because *SourcererCC* calculates similarity by counting the number of tokens that are identical for two methods. The similarity between two methods, $M1$ and $M2$, denoted as $S(M1, M2)$, is calculated by dividing the count of identical tokens in $M1$ and $M2$ by the maximum number of tokens present in either $M1$ or $M2$. Due to the consideration of only token-level information in the analysis, semantic clones are not detected, leading to a low recall. *RtvNN* has high precision but low recall. According to findings from Zhao et al. [6], the computed distances using *RtvNN* is at least 2.0 and at most 2.8. When the distance threshold is reduced to achieve a precision of 90%, the recall rapidly decreases to less than 10%. Therefore, *RtvNN* cannot have high recall and precision at the same time.

For **Tree-based** approaches, *Deckard* parses the program to obtain AST, and then the feature vectors of each subtree within the AST are clustered using predefined rules to identify code clones. However, in the process of generating abstract syntax trees by the parser, we found that over 50% of the code clone pairs cannot be detected due to different tree structures, so the recall is low. Among the tree-based comparison tools, *CDLH* is another tool that is not very effective in detecting semantic clones. That's because it utilizes an LSTM network, which is trained to learn representations encompassing hash functions, structural information, and code fragments, which are lexical and syntactic features of the code, so the effect of detecting semantic clones is not ideal. In contrast, *TBCNN*

TABLE I
THE RESULTS OF CLONE DETECTION CONDUCTED ON GCJ AND BCB DATASETS

Group	Method	GCJ			BCB		
		R	P	F1	R	P	F1
Token	SourcererCC	0.11	0.43	0.17	0.07	0.98	0.14
	RtvNN	0.90	0.20	0.33	0.01	0.95	0.01
Graph	SCDetector	0.87	0.81	0.82	0.92	0.97	0.94
	DeepSim	0.82	0.71	0.76	**0.98**	0.97	0.98
	FCCA	0.90	**0.95**	0.92	0.92	0.98	0.95
Tree	Deckard	0.44	0.45	0.44	0.06	0.93	0.12
	ASTNN	0.87	0.95	0.91	0.94	0.92	0.93
	TBCNN	0.89	0.91	0.90	0.81	0.90	0.85
	CDLH	0.70	0.46	0.55	0.74	0.92	0.82
	Amain	0.91	0.93	0.92	0.97	0.99	0.98
Our tool	Goner	**0.92**	0.92	**0.92**	0.97	**0.99**	**0.98**

and *Amain* have a better ability to detect semantic clones. The effectiveness of *TBCNN* stems from the use of sliding convolution kernels to capture the structural features of ASTs, but limited by the problem of missing contextual information over long distances and the long-term dependence of the original semantics of the source code, *TBCNN* does not have the best detection results. *Amain* possesses the capability to detect semantic clones by converting the original AST into a Markov chain model to extract the structural information in the AST, and the use of machine learning algorithms also improves the detection capability of *Amain*. *ASTNN* constructs ASTs of code fragments and splits the entire AST into small statement trees. Then a recursive encoder is designed to capture utterance-level lexical and syntactic information and represent them as utterance vectors. In addition, the use of neural networks gives *ASTNN* a high precision, however, AST segmentation brings the disadvantage of a bit low recall.

For **Graph-based** approaches, *SCDetector* analyzes the centrality of individual basic blocks within the control flow graph to preserve semantic details, however the detection effectiveness diminishes due to the decrease in the number of shared tokens caused by variations in the employed APIs and discrepancies in graph structures. *DeepSim* uses binary matrices to represent variables, basic code blocks, and the relationships between them. The consideration of the semantics of the method and the use of deep learning models enable *DeepSim* to achieve satisfactory performance on semantic clone detection. *FCCA* extracts both tokens, which are unstructured representations, and the AST and CFG, which are structured representations of the code. By combining the comprehensive hybrid code representation with a deep learning model featuring the attention mechanism, *FCCA* is equipped to identify most semantic code clones with the highest precision.

Overall, *Goner* outperforms other code clone detectors, and is excellent at detecting most of the semantic code clone pairs within the Google Code Jam dataset, and has a good ability to detect semantic clones.

2) Results on BCB

Then, we analyze the F1 score, precision, and recall when detecting all types of clones on BCB, the comparative detection results of ten comparative systems and *Goner* are shown in Table I.

From the results, it can be seen that *Goner* outperforms all our benchmark code clone detectors in both precision and F1 score. Such results indicate that *Goner* has an excellent ability to detect code clones on the BCB dataset. In addition, compared to the detection results on the GCJ dataset, the detection precision of almost all comparative systems has improved. This is because the clone pairs on the GCJ dataset are all semantic clones, and methods that only rely on the syntax information of the code will not detect semantic clones in GCJ well. It is also possible that there are two programs in the dataset that are syntactically similar but implement completely different functionality, resulting in low precision.

TABLE II
THE F1, PRECISION, AND RECALL IN DETECTING EACH TYPE OF CLONE OF *Goner*

Metrics	T1	T2	ST3	MT3	WT3/T4
F1	1.00	1.00	1.00	0.99	0.96
Precision	1.00	1.00	1.00	1.00	0.99
Recall	1.00	1.00	1.00	0.99	0.93

Moving forward, we conduct an analysis of the F1 score, recall, and precision for each of the five types and compare their F1 score with advanced code clone detector, and the comparison results are presented in Table II. Notably, the performance of *Goner* is exceptional across all three metrics for detecting T1, T2, ST3, and MT3, with scores exceeding 99%. In the case of detecting WT3/T4, *Goner* attains a remarkable F1 score of 96%, precision of 99%, and recall of 93%. This result proves that *Goner*'s processing of program semantic information is effective.

Table III shows the F1 score of different code clone detectors in the detection of five distinct types of code clones. It can be seen that *Goner* surpasses our baseline comparison technique in detecting each of the different types of code clones. Notably, when it comes to the detection of WT3/T4 code clones, the F1 scores achieved by *SourcererCC*, *RtvNN*, *Deckard*, *ASTNN*, *TBCNN*, and *CDLH* are 2%, 0%, 2%, 92%, 86%, and 82% respectively, *Goner* showcases a higher F1 score of 96%. Compared with these tools, *Goner* greatly improves the efficiency of detecting WT3/T4 clones. It shows the effectiveness and superiority of *Goner* in processing program semantic information and detecting semantic clones, and *Goner* detects code clones well. Compared with *SCDetector*, *DeepSim*, and *FCCA*, which can also detect WT3/T4 clones with good performance, *Goner* is more scalable than them. Because they all need GPU to complete the training phase of the deep neural network, and *Goner* only needs the CPU to complete the training of the machine learning model, in other words, *Goner* requires fewer computing resources. Compared with *Amain*, *Goner* also performs slightly better when detecting semantic clones.

The answer to RQ2: Compared with other code clone detectors, *Goner* is excellent in detecting code clones on both GCJ dataset and BCB dataset, especially in detecting semantic clones.

D. RQ3: The Significance of Position

To illustrate that our algorithm for computing similarity features based on position information of subtrees is effective

TABLE III
F1 FOR EACH CLONE TYPE ON BCB

Group	Method	T1	T2	ST3	MT3	T4
Token	SourcererCC	1.00	1.00	0.65	0.20	0.02
	RtvNN	1.00	0.97	0.6	0.03	0.00
Graph	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
Tree	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
	Amain	1.00	1.00	0.99	0.99	0.95
Our tool	Goner	**1.00**	**1.00**	**1.00**	**0.99**	**0.96**

for clone detection, we implement an ablation experiment on BCB and GCJ datasets using 2-gram. Specifically, we input the similarity vectors obtained in the feature extraction phase by considering (*i.e.,* which we called *Goner* in Table IV) as well as not considering (*i.e.,* which we called *Goner-* in Table IV) position information into the RF machine learning algorithm for training and testing, respectively. We record their respective detection results in Table IV.

TABLE IV
RECALL, PRECISION, AND F1 OF GONER CONSIDERING AND NOT CONSIDERING POSITION INFORMATION IN DETECTING CLONES

Dataset	Method	Dimension	R	P	F1
GCJ	Goner	107	0.92	0.92	0.92
	Goner-	58	0.88	0.89	0.89
BCB	Goner	116	0.97	0.99	0.98
	Goner-	63	0.96	0.98	0.97

From the experimental results recorded in the table, it can be seen that on the two datasets, the consideration of the node position information in the subtree has a positive effect on the detection of code clones. For example, on the GCJ dataset, the F1 score is 91.94% when we consider the location information to extract features, and only 89.05% when the location information is not considered, a decrease of nearly 3%. We believe that the position information of the nodes in the subtree can largely reflect the structure of the AST, and the different positional relationships of the two nodes can reflect the different logical structures of the program, thus containing different program information. Therefore, the method that considers positional information can achieve better detection results. At the same time, we exclude the effect of decreasing the dimension of the similarity vector on the detection effect of *Goner-*. From the results in Section IV-G, it can be known that even if only the first 58-dimensional vectors of *Goner* are retained, better results than *Goner-* can be obtained. It can be explained that, compared with *Goner-*, *Goner* retains more semantic information by analyzing the position information of nodes, so it has a better detection effect.

The answer to RQ3: The consideration for location information in the feature extraction phase does have a positive effect on detection.

E. RQ4: The Significance of Using Machine Learning

To illustrate the improvement in the effectiveness of using machine learning algorithms for clone detection, this part compares it to detecting clones using a simple threshold-based approach. We use 2-gram on the GCJ dataset to extract vectors of a pair of methods, and then calculate the cosine similarity between these two vectors. If the similarity result is larger than the threshold, we will report them as a clone pair, otherwise, they will be considered as a non-clone pair.

TABLE V
RECALL, PRECISION, AND F1 OF GONER BY USING SIMPLE THRESHOLD-BASED APPROACH AND MACHINE LEARNING ALGORITHMS IN DETECTING CLONES

Threshold / ML Method	Recall	Precision	F1
0.6	0.99	0.51	0.67
0.65	0.99	0.51	0.67
0.7	0.98	0.51	0.67
0.75	0.96	0.52	0.68
0.8	**0.92**	**0.54**	**0.68**
0.85	0.85	0.57	0.68
0.9	0.68	0.63	0.65
RF	**0.92**	**0.92**	**0.92**

As shown in Table V, the threshold-based approach achieves the best F1 score of 0.68 when the threshold is 0.8, but it is still far below the detection effect of 0.92 obtained by using the RF algorithm. This result well reflects the excellent superiority of machine learning algorithms in classification, and the use of machine learning algorithms can indeed improve the detection effect of code clones.

The answer to RQ4: The use of machine learning algorithms does have a positive effect on clone detection.

F. RQ5: Scalability

In this subsection, we measure the scalability of *Goner* by analyzing the runtime overhead of four phases. From subsection IV-B, we know that ideal detection results can be obtained when using 2-gram and the Random Forest algorithm. So we record the running overheads of the 2-gram and Random Forest algorithm.

To complete the runtime overhead comparison, we initially select one million code pairs randomly from the GCJ dataset to serve as our test data. It was recorded that it took only 2.5 seconds to generate features for one million clone pairs. Next, we compare the runtime performance of *Goner* and the other ten systems. In Table VI, we give the training and testing overheads of our comparative systems and *Goner*. The training time includes both the training time for machine learning and the time of previous data processing, including the generation and segmentation of the AST and the extraction of features.

The training overheads of *SourcererCC* and *Deckard* are both zero because they have no training phase. *SourcererCC* takes a small number of time to detect clones because it only analyzes the source code tokens. *Deckard* also takes little time to detect clones, but as shown in Tables I and Table III, its detection performance is not good. *RtvNN* has no advantage in running overhead, and its ability to detect semantic clones is notably limited. In the case of *DeepSim*, *FCCA*, *ASTNN*, *TBCNN*, and *CDLH*, these methods took a super long time to complete the training phase, and even though their prediction phase didn't take an exaggeratedly long time, it still took longer than *Goner*. For *SCDetector*

TABLE VI
RUNTIME ON ANALYZING ONE MILLION CODE PAIRS

Group	Method	Training	Prediction
Token-based	SourcererCC	-	16s
	RtvNN	5,206s	35s
Graph-based	SCDetector	2,937s	139s
	DeepSim	13,545s	34s
	FCCA	56,769s	91s
Tree-based	Deckard	-	72s
	ASTNN	16,096s	2,894s
	TBCNN	41,168s	86s
	CDLH	45,317s	90s
	Amain	1,017s	32s
Our tool	Goner	311s	24s

TABLE VII
TOP TEN FEATURES OF *Goner* IN DETECTING SEMANTIC CODE CLONES

Rank	Feature Name	Weight
1	ForStatement_child	0.070754
2	StatementExpression_parent	0.063626
3	BinaryOperation_child	0.060112
4	BasicType_child	0.049270
5	ArrayCreator_parent	0.042091
6	BinaryOperation_parent	0.037299
7	MemberReference_parent	0.035472
8	Literal_child	0.034041
9	BlockStatement_parent	0.033277
10	VariableDeclarator_parent	0.032617

and *Amain*, although they save a lot of time compared to the above five tools, they still take more time overhead (*i.e.,* 3,076 seconds and 1,041 seconds) than *Goner*. For *Goner*, we only need to spend about 335 seconds in total (*i.e.,* 311 seconds for training and 24 seconds for testing) for one million clone-pair samples detection. In other words, *Goner* is about 56 times (*i.e.,* 18,990/335=56.7) faster than *ASTNN*, nine (*i.e.,* 3,076/335=9.2) times faster than *SCDetector*, and three (*i.e.,* 1,041/335=3.1) times faster than *Amain*.

In summary, *Goner* has a larger runtime overhead than the token-based approach (*i.e., SourcererCC*) because *Goner* incorporates the semantics of the program into its analysis and detection process. However, due to the utilization of machine learning techniques and n-gram variants to segment the original abstract syntax tree, *Goner* has the capability to detect semantic code clone. In comparison to *ASTNN*, *Goner* is about 51 times (*i.e.,* 16,096/311=51.7) faster in training phase and 120 times (*i.e.,* 2,894/24=120.6) faster in predicting phase.

The answer to RQ5: *Goner* only takes about 335 seconds in total to detect one million clone pairs, it takes less time than most comparison tools.

G. RQ6: Interpretability

To explore in more depth why *Goner* has such an excellent performance in detecting semantic clones, in this part, we analyze the importance of each feature of *Goner*. The advantage of using the RF algorithm lies in its interpretability, as it enables us to gather the importance values of all the 107 features employed in the model. We rank these features by their weights, and then identify the features that carry more significance in the detection process. Due to space constraints, we can only present the top ten features in Table VII. Among them, "_child" indicates that the node of this type is in the position of the child node, and "_parent" indicates that the node of this type is in the position of the parent node.

By observing the ranking of feature weights, we find two remarkable phenomena. First, the node types that appear in the top ten retain more semantic information and are therefore more important. For example, *ForStatement* connects statements in the body of a *for* loop, these statements are often a concentrated representation of the functionality implemented by a method, containing a wealth of semantic information. *BinaryOperation* represents a binary operation (*e.g.,* "`a<=0`"), which is usually found in conditional judgments such as *If* statements, *For* and *While* conditional judgment statements. These statements are most likely to be a concentration of method functionality, with control flow information and rich semantics. Therefore *BinaryOperation* is important for the representation of semantics in methods as it connects different statements. The nodes connected by *ArrayCreator* can reflect the structure information related to the array such as the array's type and size, such as "`new double[5]`". The same data structure can reflect the same functionality, so the relevant information related to arrays plays a crucial role in detection.

Second, nodes of the same type will have different levels of importance due to their different locations. For example, *ForStatement* is ranked first when it is in the child position and 31st when it is in the parent position. This is because *ForStatement* represents the statements related to the *For* loop. When *ForStatement* is a child node, its parent node is usually the outer nest, which reflects the structure of the method and thus contains semantic information. When *ForStatement* is a parent node, the connected child nodes are statements within the loop body, which will change according to different loop requirements. Therefore, *ForStatement_parent* is not as important as *ForStatement_child*. *StatementExpression* is ranked second when it is in the parent position and 21st when it is in the child position. This is because *StatementExpression* can point to different statement types, For example, when the statement is an assignment statement, the child node of *StatementExpression* is *Assignment*, and when the statement is a function call, the child node is *MethodInvocation*. Therefore, the child nodes connected by *StatementExpression* in the abstract syntax tree (AST) can indeed reflect the semantic information of the method. When *StatementExpression* is a child node, its parent node is usually *BlockStatement*, which represents a block of statements such as function body, which is not helpful for the embodiment of method semantics. Therefore, *StatementExpression_child* is not as important as *StatementExpression_parent*.

To analyze the contribution of these features more clearly, we construct similarity vectors of varying lengths (ranging from 1 to 107) by selecting the top n features in descending order of importance. Then the F1 scores for clone detection using similarity vectors of different lengths are recorded in Figure 12. We find that using only one feature for classification can achieve an F1 score of 0.7074. With the increasing number of features, the detecting effect becomes better and better. When the vector length reaches about 50, the ideal F1 score

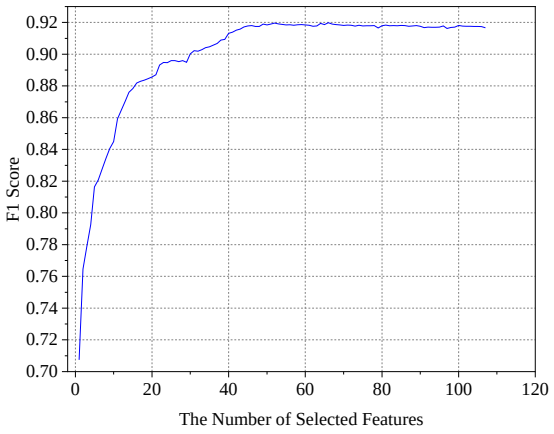


Fig. 12. F1 scores of *Tritor* when selecting different numbers of features

(0.92) can already be basically achieved, which means that only the first 50 important features can enable *Goner* to have excellent semantic clone detection capabilities.

The answer to RQ6: The extraction of features with strong semantic properties gives *Goner* the ability to detect semantic clones.

V. Discussions

Why our method is called N-gram-like model. Our method is derived from the traditional N-gram method which is used to segment strings. Therefore, there are both differences and similarities between them. The difference is that the division unit of the traditional n-gram method is token, while the unit of our tree-based N-gram-like model is the node in the AST. Although the N-gram method and our N-gram-like model are different in form, they share the same division principle. They both start from the starting position, divide the connected n units into groups, and traverse to the last unit to complete the division. Our method is inspired by the traditional N-gram method but differs from it. Therefore, our method is named as Tree-based N-gram-like model.

Why Goner is superior to other approaches. First, the extraction of the intermediate representation of the program, the abstract syntax tree, enables *Goner* to process the semantic information of the program efficiently. Thus, *Goner* has a good ability to detect semantic clones. Second, the use of N-gram-like model to partition the original abstract syntax tree makes it easy and convenient for *Goner* to preserve the syntactic information embedded in the AST without dealing with the complex tree structure. Third, we focus on classifying each subtree according to its positional information, so the formed features are rich in positional information, which in turn carry and preserve the semantic features of the program to a greater extent. Fourth, compared to calculating the similarity between code pairs directly and determining whether a code pair is a clone through a threshold, we put the similarity between code pairs into the machine learning model, which greatly enhances the accuracy of clone detection. In fact, we have tried to use N-gram-like to segment the AST to obtain the features and directly calculate the similarity of the two programs to determine whether the code pair is a clone by setting a threshold. The accuracy of this approach is very bad, demonstrating the effectiveness of classification based on location information, as well as the effectiveness of machine learning algorithms for classification.

Threats to Validity. The first threat comes from the dataset we used to verify validity and scalability. The code pairs in the BCB dataset, a widely used benchmark dataset for code clone detection, are manually constructed and classified by several experts. Differences in code pairs and changes in their structure are small and can be easily detected. Therefore, the results obtained by using only the BCB dataset for validation are biased and not representative of the entire open source project. To mitigate the impact of the dataset, we added experiments conducted on the GCJ dataset, which was written by participating programmers in 12 different competition problems containing 1,669 projects, closer to the real-world code situation.

The second threat comes from the logging of time overhead. Different machine states, such as CPU usage, can affect the runtime results. The time overhead varies across real-world scenarios, so we cannot obtain absolutely accurate and generalized data. To mitigate this threat, we ran the experiment ten times and recorded the time overhead for each time. The average time overhead for ten runs is reported in the paper, rendering our results more generalizable.

Future work. This system has universal applicability in programming languages. While our experiments primarily focused on *Java* code, it is important to note that extending the capabilities of our approach to code written in languages other than *Java* is feasible with some modifications. Indeed, it can be adapted to other languages by simply changing its static analysis tool according to the programming language. In addition, through the observations from Figure 12, we find that the top 50 features can achieve the ideal detection effect. Therefore, we may use different feature selection techniques in the future to make further trade-offs to make *Goner* more scalable while maintaining the existing good detection effect.

VI. Related Work

We focus on the current researches about clone code detection in this section. According to the different representations generated by the source code, existing clone code detection methods can be divided into five categories, text-based, token-based, tree-based, graph-based and metric-based.

For text-based clone detection techniques [33]–[40], these techniques compare the similarity of two codes by considering the source code as a series of lines or strings. Johnson [33] extracts fingerprints on the source code and then matches the corresponding fingerprints to detect clones. Roy et al. [35] first normalizes the clone and then matches the longest common subsequence of the code text. These methods do not consider the syntax and semantics of the program and only detect those clones with high text similarity by string matching, making it difficult to find semantic clones.

For token-based clone code detection techniques [1], [2], [41]–[45], these methods perform lexical analysis of the program code to convert the source code into token sequences and

detect code clones by finding duplicate token subsequences. *SourcererCC* [2] calculates similarity by counting the number of tokens that are identical for two methods. However, the approach based on token cannot detect semantic clones.

For tree-based clone code detection techniques [8]–[12], [46], [47], adopt a parsing process to transform the program into a tree structure, capturing its semantic information, and then perform detection using tree matching. *Deckard* [8] can use *locality sensitive hashing* (LSH) to cluster similarity vectors within the AST to detect clones for any grammatically specified language. *CDLH* [9] first use a binary tree instead of the original AST, and then uses Tree-LSTM [48] to encode the tree into a vector to measure similarity. In the case of *ASTNN* [10], the abstract syntax tree (AST) is divided into multiple subtrees according to predefined rules, and uses a bidirectional RNN model to integrate the vectors encoded by the subtrees into the final vector representation. Jo et al. [12] using a *tree-based convolutional neural network* (TBCNN) to perform clone detection. The tools based on tree are generally good at detecting semantic clones. However, tree processing and tree matching take a long time, so the scalability is poor.

Graph-based code clone detection techniques [3]–[7], [19], adopt a parsing process to convert the program into various graph representations, just like PDG or CFG, these techniques utilize subgraph matching algorithms [3], [4] to detect code clones. For *DeepSim* [6], the control flow and data flow of the code are transformed into semantic representations encoded as high-dimensional sparse matrices, which are converted into a classification problem of binary feature vectors to detect code clones. *CCGraph* [7] filters the reduced PDG using two steps and finally uses an approximate graph matching algorithm to identify the clone pairs. For *SCDetector* [19], the CFG is treated as a social network, and the concept of centrality is applied to each block within the graph. Similar to the tree-based approach, the use of graph isomorphism and graph matching makes the graph-based approach poorly scalable.

Metric-based methods [49]–[57] exploit the properties of the code to assess the degree of similarity. Metrics can be obtained from intermediate representations such as tree or graph, for example, Balazinska et al. [50] use AST as metric to detect code clones. Metrics can also be directly derived from code itself, Patenaude et al. [51] detect clones on metrics of different categories (*e.g.,* classes, hierarchies, and couplings). *Oreo* [52] has high precision and recall because it combines machine learning, software metrics, and information retrieval.

The methods based on text and token have the least runtime overhead, but cannot detect semantic clone. Graph-based methods have the capability to detect semantic clones, but obtaining the graph structure typically requires the code to be compiled, which cannot be carried out on some fine-grained code fragments. The tree-based approach is being widely used because it can preserve the semantic information without compiling it. Even for a small code snippet, the resulting AST can be quite intricate. So there are some hybrid code representation methods to detect clones that have good scalability while ensuring accuracy. For example, *FCCA* [20] selects multiple code representations, including ASTs, CFGs, and sequence tokens, to fuse into a hybrid representation, an attention mechanism was employed to identify code clones.

VII. Conclusion

This paper presents a novel and scalable AST-based approach namely *Goner* for detecting semantic code clones. To avoid heavy-weight tree matching, *Goner* builds N-gram-like model to segment the original AST, and generates features based on the location information of subtrees with the aim of efficient code clone detection. We test *Goner* on the BigCloneBench dataset and Google Code Jam dataset. Based on the experimental results, it is evident that *Goner* surpasses ten current state-of-the-art code clone detectors in terms of the effectiveness of detecting semantic clones. Compared to another state-of-the-art tree-based semantic code clone detector (*i.e., ASTNN*) in terms of scalability, *Goner* is about 51 times faster in training phase and 120 times faster in predicting phase.

ACKNOWLEDGEMENTS

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the Key Program of the National Science Foundation of China under Grant No. U1936211 and Hubei Key Project under Grant No. 2023BAA024.

References

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[2] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big code," in *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, 2016, pp. 1157–1168.

[3] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, 2001, pp. 301–309.

[4] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 2001 International Static Analysis Symposium (ISAS'01)*, 2001, pp. 40–56.

[5] M. Wang, P. Wang, and Y. Xu, "Ccsharp: An efficient three-phase code clone detector using modified pdgs," in *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*, 2017, pp. 100–109.

[6] G. Zhao and J. Huang, "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)*, 2018, pp. 141–151.

[7] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "Ccgraph: a pdg-based code clone detector with approximate graph matching," in *Proceedings of the 35th International Conference on Automated Software Engineering (ASE'20)*, 2020, pp. 931–942.

[8] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 96–105.

[9] H. Wei and M. Li, "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)*, 2017, pp. 3034–3040.

[10] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*, 2019, pp. 783–794.

[11] H. Liang and L. Ai, "Ast-path based compare-aggregate network for code clone detection," in *Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN'21)*, 2021.

[12] Y. B. Jo, J. Lee, and C. J. Yoo, "Two-pass technique for clone detection and type classification using tree-based convolution neural network," *Applied Sciences-Basel*, vol. 11, no. 14, 2021.

[13] "Bigclonebench," https://github.com/clonebench/BigCloneBench, 2023.

[14] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "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)*, 2014, pp. 476–480.

[15] "Google code jam," https://code.google.com/codejam/past-contests, 2017.

[16] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 87–98.

[17] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI'16)*, 2016, pp. 1287–1293.

[18] Y. Wu, S. Feng, D. Zou, and H. Jin, "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)*, 2022, pp. 1–13.

[19] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, "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)*, 2020.

[20] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2021.

[21] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[22] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[23] "A pure python library for working with java source code, provies a lexer and parser targeting java 8. (javalang)," https://pypi.org/project/javalang/, 2023.

[24] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[25] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[26] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[27] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of online learning and an application to boosting," *Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[28] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of Statistics*, pp. 1189–1232, 2001.

[29] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[30] "An open source machine learning library that supports supervised and unsupervised learning. (scikit-learn)," https://scikit-learn.org/stable/, 2023.

[31] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *Proceedings of the 15th International Conference on Machine Learning and Applications (ICMLA'16)*, 2016, pp. 1024–1028.

[32] Y. Hu, D. Zou, J. Peng, Y. Wu, J. Shan, and H. Jin, "Treecen: Building tree graph for scalable semantic code clone detection," in *Proceedings of the 37th International Conference on Automated Software Engineering (ASE'22)*, 2022, pp. 1–12.

[33] J. H. Johnson, "Substring matching for clone detection and change tracking," in *Proceedings of the 1994 International Conference on Software Maintenance (ICSM'94)*, 1994, pp. 120–126.

[34] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the 1999 International Conference on Software Maintenance (ICSM'99)*, 1999, pp. 109–118.

[35] C. K. Roy and J. R. Cordy, "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)*, 2008, pp. 172–181.

[36] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *Proceedings of the 11th International Workshop on Software Clones (IWSC'17)*, 2017, pp. 8–14.

[37] S. Kim and H. Lee, "Software systems at risk: An empirical study of cloned vulnerabilities in practice," *Computers Security*, vol. 77, pp. 720–736, 2018.

[38] S. Jadon, "Code clones detection using machine learning technique: Support vector machine," in *Proceedings of the 2016 International Conference on Computing, Communication, and Automation (ICCCA'16)*, 2016, pp. 299–303.

[39] D. Yu, J. Wang, Q. Wu, J. Yang, J. Wang, W. Yang, and W. Yan, "Detecting java code clones with multi-granularities based on bytecode," in *Proceedings of the 41st Annual Computer Software and Applications Conference (COMPSAC'17)*, 2017, pp. 317–326.

[40] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 38th Symposium on Security and Privacy (SP'17)*, 2017, pp. 595–614.

[41] N. Göde and R. Koschke, "Incremental clone detection," in *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (ECSMR'09)*, 2009, pp. 219–228.

[42] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *Proceedings of the 2017 International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017, pp. 249–260.

[43] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: A token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018, pp. 1066–1077.

[44] Y. Golubev, V. Poletansky, N. Povarov, and T. Bryksin, "Multi-threshold token-based code clone detection," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*, 2021.

[45] Y. L. Hung and S. Takada, "Cppcd: A token-based approach to detecting potential clones," in *Proceedings of the 14th IEEE International Workshop on Software Clones (IWSC'20)*, 2020.

[46] J. Pati, B. Kumar, D. Manjhi, and K. K. Shukla, "A comparison among arima, bp-nn, and moga-nn for software clone evolution prediction," *IEEE ACCESS*, vol. 5, pp. 11 841–11 851, 2017.

[47] S. Chodarev, E. Pietrikova, and J. Kollar, "Haskell clone detection using pattern comparing algorithm," in *Proceedings of the 13th International Conference on Engineering of Modern Electric Systems (EMES'15)*, 2015.

[48] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[49] J. Mayrand, C. Leblanc, and E. Merlo, "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)*, 1996, pp. 244–253.

[50] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Proceedings of the 6th International Software Metrics Symposium (ISMS'99)*, 1999, pp. 292–303.

[51] J. F. Patenaude, E. Merlo, M. Dagenais, and B. Laguë, "Extending software quality assessment techniques to java systems," in *Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99)*, 1999, pp. 49–56.

[52] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "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)*, 2018, pp. 354–365.

[53] C. Ragkhitwetsagul, J. Krinke, and B. Marnette, "A picture is worth a thousand words: Code clone detection based on image similarity," in *Proceedings of the 12th International Workshop on Software Clones (IWSC'18)*, 2018, pp. 44–50.

[54] S. M. F. Haque, V. Srikanth, and E. S. Reddy, "Generic code cloning method for detection of clone code in software development," in *Proceedings of the 2016 International Conference on Data Mining and Advanced Computing (SAPIENCE'16)*, 2016, pp. 340–344.

[55] S. M and L. Rangarajan, "Code clone detection based on order and content of control statements," in *Proceedings of the 2nd International Conference on Contemporary Computing and Informatics (IC3I'16)*, 2016, pp. 59–64.

[56] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE'17)*, 2017, pp. 27–30.

[57] M. Tsunoda, Y. Kamei, and A. Sawada, "Assessing the differences of clone detection methods used in the fault-prone module prediction," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, 2016, pp. 15–16.