Detecting Semantic Code Clones by Building AST-based Markov Chains Model

Yueming Wu[*†]
Huazhong University of Science and Technology
China
wuyueming21@gmail.com

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

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

Hai Jin[†§]
Huazhong University of Science and Technology
China
hjin@hust.edu.cn

ABSTRACT

Code clone detection aims to find functionally similar code fragments, which is becoming more and more important in the field of software engineering. Many code clone detection methods have been proposed, among which tree-based methods are able to handle semantic code clones. However, these methods are difficult to scale to big code due to the complexity of tree structures. In this paper, we design *Amain*, a scalable tree-based semantic code clone detector by building Markov chains models. Specifically, we propose a novel method to transform the original complex tree into simple Markov chains and measure the distance of all states in these chains. After obtaining all distance values, we feed them into a machine learning classifier to train a code clone detector. To examine the effectiveness of *Amain*, we evaluate it on two widely used datasets namely Google Code Jam and BigCloneBench. Experimental results show that *Amain* is superior to nine state-of-the-art code clone detection tools (*i.e., SourcererCC*, *RtvNN*, *Deckard*, *ASTNN*, *TBCNN*, *CDLH*, *FCCA*, *DeepSim*, and *SCDetector*).

CCS Concepts

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

Keywords

Semantic Code Clones, Abstract Syntax Tree, Markov Chain

[*]Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
[†]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, HUST, Wuhan, 430074, China
[‡]Deqing Zou is the corresponding author
[§]Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3560426

ACM Reference Format:
Yueming Wu, Siyue Feng, Deqing Zou, and Hai Jin. 2022. Detecting Semantic Code Clones by Building AST-based Markov Chains Model. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3560426

1 INTRODUCTION

On one hand, the continuous development of the software field makes the scale of software code larger and larger. On the other hand, since code cloning can save developers' time and effort, developers are inclined to clone code with similar functionalities rather than implement them from scratch. In fact, code cloning brings convenience but also increases maintenance costs. For example, if there are vulnerable codes in the source code, their code clones can lead to vulnerability propagation, requiring maintainers to perform vulnerability scanning on more projects to ensure software security. To mitigate the situation, code clone detection has become an active area of research and is increasingly important to software engineers.

There have been proposed many code clone detection methods. For example, *CCFinder* [20] first extracts token sequences for program code and then develops transformation rules to transform the token sequences, which can be used to detect Type-1 and Type-2 clones. *SourcererCC* [34] first applies lexical analysis to obtain the tokens and then compares the overlapping parts of these tokens to detect Type-3 clones. However, because they do not take into account the semantics of the program, they are not able to detect Type-4 clones. To tackle the problem, researchers propose to extract the intermediate representations of the program to maintain its semantics. For example, some methods [21, 22, 41, 49, 50] transform the program details into a graph representation (*e.g.,* program dependency graph) and perform graph analysis to achieve accurate semantic code clone detection. However, graph analysis is time-consuming which makes it hard to scale to big code. To mitigate the issue, other approaches [17, 18, 24, 43, 48] prefer to obtain the tree representation (*e.g.,* abstract syntax tree) and apply tree matching to detect semantic code clones. However, although tree parsing is more lightweight than graph analysis, the tree structures are still complex. For example, Figure 2(a) shows the abstract syntax tree corresponding to the boxed part of the original in Figure 1. As we can see from the figure, a simple two lines of code can possess a

complex subtree with 26 nodes. When a method has more lines of code, the corresponding tree will be more complex, resulting in a high overhead for tree analysis. Such high overhead suggests that they are not suitable for daily large-scale code clone scanning. As a result, we need to devise a method that can effectively reduce the high overhead imposed by tree analysis.

In this paper, we design a novel scalable tree-based semantic code clone detector to assist large-scale code clone detection. Specifically, we mainly address two challenges:

- *Challenge 1: How to parse the original complex tree into simple subtrees while maintaining the tree details?*
- *Challenge 2: How to design a succinct yet effective code clone detection process to deal with semantic code clones?*

To address the first challenge, we build a Markov chains model to represent the complex *abstract syntax tree* (AST). Markov chains are often used in the field of probability and statistics to describe the probability of transitioning from one state to another. In our approach, we consider the AST as a state transfer graph in a Markov chain, and the pointing from parent to child nodes as a transfer from one state to another. In this way, the tree structure that represents the program logic is transformed into certain transfers between different states. The transfer probability from one state to another state carries the tree details of the program. Based on the built model, we can achieve scalable tree analysis while preserving the program details.

To solve the second challenge, we compute different distance values between all states and construct the corresponding feature vectors to train a code clone detector. Specifically, we select four widely used distance measures (*i.e., Cosine distance*, *Euclidean distance*, *Manhattan distance*, and *Chebyshev distance*) to achieve comprehensive distance computation of the same state in different programs. After collecting the distance scores of all states, we use them to train a machine learning classifier for code clone detection. Based on the trained classifier, we can achieve simple and effective semantic code clone prediction.

We implement a prototype system, *Amain*, and evaluate it on two widely used datasets namely Google Code Jam [1] and BigCloneBench [2, 36]. Our experiments show that *Amain* can achieve better detection performance than nine state-of-the-art code clone detection systems including two token-based methods (*i.e., SourcererCC* [34] and *RtvNN* [44]), four tree-based methods (*i.e., Deckard* [17], *ASTNN* [48], *TBCNN* [26], and *CDLH* [43]), and three graph-based methods (*i.e., SCDetector* [46], *DeepSim* [49], and *FCCA* [14]). In addition, we also compare the scalability of *Amain* with these nine systems. Experimental results report that although *Amain* takes more time than token-based approaches (*e.g., SourcererCC* [34]), it is 15 times faster than another state-of-the-art tree-based approach (*i.e., ASTNN* [48]) in the training phase and 160 times faster in the testing phase.

In summary, this paper makes the following contributions:

- We propose a novel method to convert the complex AST into simple Markov chains and extract features by measuring distances from the generated transfer probability matrix.
- We design a prototype system namely *Amain*[1] by building a Markov chains model and training a machine learning classifier. The built model and learned classifier enable us to achieve scalable yet accurate semantic code clone detection.
- We perform comparative evaluations with nine systems on Google Code Jam [1] and BigCloneBench [2, 36] datasets. Experimental results show that *Amain* has the best detection performance over *SourcererCC* [34], *RtvNN* [44], *Deckard* [17], *ASTNN* [48], *TBCNN* [26], *CDLH* [43], *SCDetector* [46], *DeepSim* [49], and *FCCA* [14].

[1]https://github.com/CGCL-codes/Amain.

2 MOTIVATION

To illustrate how our approach is proposed and how semantic clones are detected, we use a simple but clear example. The original method and the Type-4 method in Figure 1 are a semantic clone pair that implements the functionality of computing the factorial of a number with different syntax. Since the two methods differ almost only in the boxed part and the complete AST is relatively large, we only calculate the similarity of the code in the boxed part.

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

```
private long factorial(long n){
    long s = 1;
    int j=1;
    while(j<=n){
        s = s*j;
        j++;}
return s;
}// Type-4
```

Figure 1: A semantic code clone pair

SourcererCC [34] is a current state-of-the-art token-based clone detector that considers only the lexical information of the methods. When calculating the similarity of two methods, *SourcererCC* divides the number of shared tokens of the two methods by the maximum of the number of tokens of the two methods to obtain the overlapping similarity. For the two methods in Figure 1, we find that the number of tokens in the boxed part are 20 and 22, respectively. After analyzing the number of shared tokens (*i.e.,* 13), we can calculate the overlapping similarity as 13/22 = 0.59. However, the default similarity threshold of *SourcererCC* is 0.7, and only code pairs with similarities exceeding the threshold can be identified as clone pairs. In other words, *SourcererCC* will flag the two methods as a non-clone pair.

To find a way to determine the two methods as a clone pair, we consider extracting their ASTs. Figure 2 shows the subtrees of the code fragments framed in Figure 1, respectively. We can see that the two code fragments share a similar tree structure although they are syntactically dissimilar. For example, the subtrees in the boxes in Figure 2 are almost identical, differing only in the leaf nodes, which represent the tokens of the code. But once we use the type of the token to represent the node (*i.e.,* the word in red indicates the type of the token), the subtrees can be identical. To better describe the tree details, we replace the leaf nodes (*i.e.,* tokens) with their corresponding types and present the number of edges between all nodes in Figure 2. After our statistical analysis, we find that the number of unique nodes in Figure 2(1) and Figure 2(2) are 14 and 13, respectively. Besides, there are 11 nodes that are the same between them. To characterize their tree details, we use a $14 * 14$ matrix to represent Figure 2(1) and a $13 * 13$ matrix to represent Figure 2(2), where the element n at position $[i, j]$ of the matrix represents that

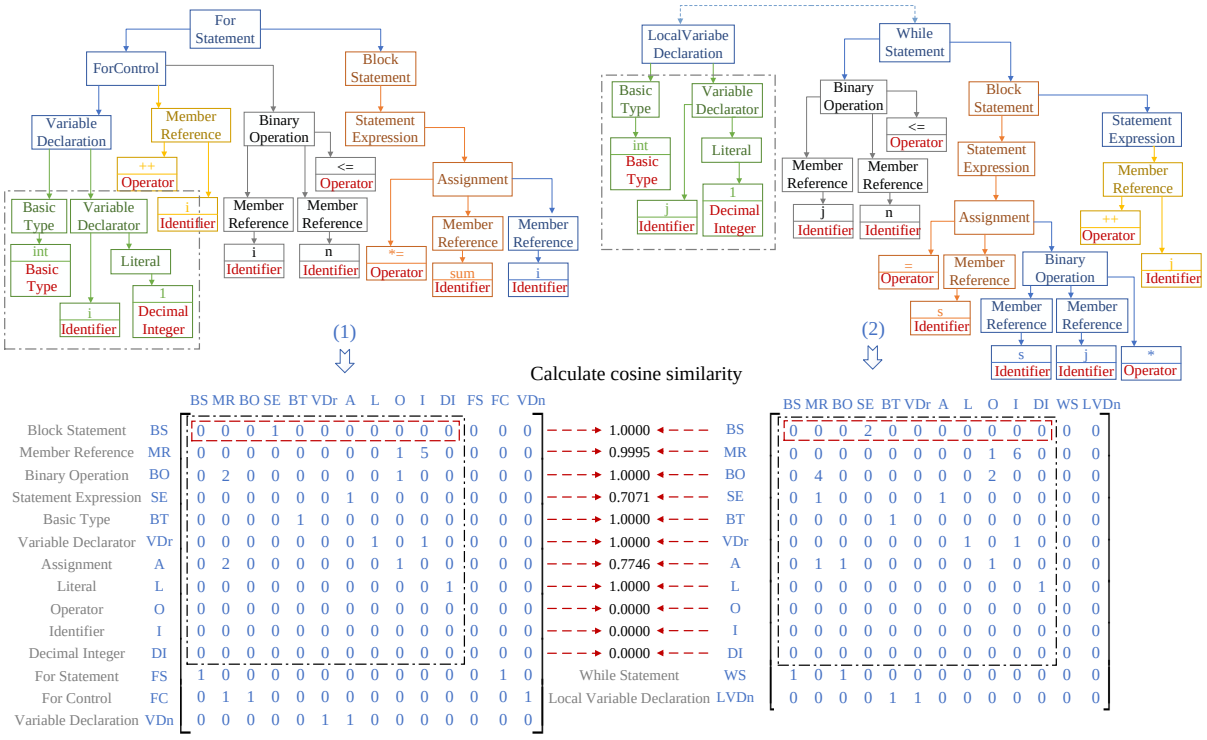

Figure 2: The AST subtree of the original method and the Type-4 method in Figure 1 and their corresponding edge matrices

there are n edges of the i_{th} node to the j_{th} node. For example, the second row of the first matrix in Figure 2 represents that there is an edge from *MemberReference* to *Operator* and there are five edges from *MemberReference* to *Identifier* in Figure 2(1).

For these two matrices, we observe that since 11 nodes are the same among them, most of the data in the matrix describe the number of edges between the same nodes. To make these data clearer, we frame them with a box. This small matrix describes the distribution of the number of edges between the 11 nodes. When we use a row of the small matrix as a unit to calculate the similarity of two small matrices, we find that the cosine similarity of different rows is not the same. Although their average similarity (*i.e.,* (1+0.9995+1+0.7071+1+1+0.7746+1+0+0+0)/11=0.68) is below 0.7, most of the rows are above 0.7. If we can assign higher weights to more similar rows, there is a good chance that the final similarity will be higher than 0.7.

Therefore, based on the observation, we build a novel technique to model all nodes in an AST and train a classifier that can assign suitable weights for different nodes to detect semantic clones.

3 CLONE TYPES

Code cloning can be classified into four types according to the degree of similarity. In our paper, we use the following definitions of code cloning types [7, 31]:

- **Type-1 (textual similarity)**: Identical code fragments, except for different white-space, layout, and comments.
- **Type-2 (lexical similarity)**: Identical code fragments, except for differences in identifier names and lexical values, in addition to the differences in Type-1 clones.
- **Type-3 (syntactic similarity)**: Syntactically similar code snippets that differ at the statement level. In addition to Type-1 and Type-2 clone differences, the fragments have statements added, modified, and/or removed with respect to each other.
- **Type-4 (semantically similarity)**: Syntactically dissimilar code fragments that implement the same functionality.

4 SYSTEM

In this section, we introduce our proposed tree-based semantic code clone detector *Amain*. To deal with the time-consuming tree matching problem, we regard the nodes of an AST as different states and build a Markov chains model to transform the complex tree into simple state transitions. After collecting the distance vectors of all states between ASTs, we can train a classifier that can assign suitable weights for different states to achieve succinct and effective semantic code clone detection.

4.1 Overview

As described in Figure 3, *Amain* is mainly comprised of four phases: *AST Generation*, *State Matrix Construction*, *Feature Extraction*, and *Classification*.

- ***AST Generation***: The purpose of this phase is to apply static analysis to generate the corresponding AST. The input of this phase is a method and the output is an AST.
- ***State Matrix Construction***: The purpose of this phase is to transform the AST into a Markov chain-based state matrix. The input of this phase is an AST and the output is a state transfer matrix.

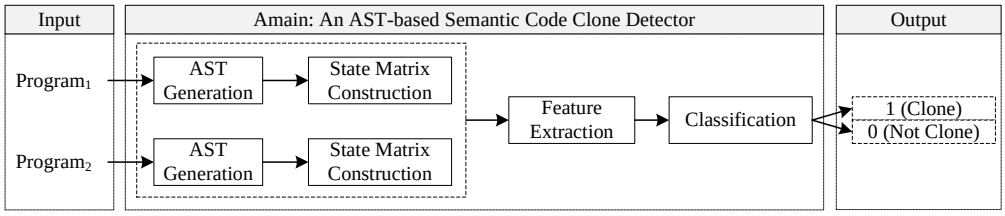

Figure 3: System architecture of *Amain*

- ***Feature Extraction***: The purpose of this phase is to calculate the distance vectors (*i.e.,* features) of two state transfer matrices. The input of this phase is two state transfer matrices and the output is a feature vector.
- ***Classification***: The purpose of this phase is to determine whether two methods are semantically similar or not. The input of this phase is a feature vector and the output reports the detection results.

4.2 AST Generation

Amain's purpose is to efficiently and effectively detect semantic code clones by converting the AST into the form of a matrix using the principle of the Markov chain. Therefore, we need to conduct static analysis to extract the AST. Since the programming language of our experimental dataset is Java, we use *Javalang* [4] to complete our static analysis.

4.3 State Matrix Construction

For the obtained AST, instead of using conventional heavyweight tree matching approaches to detect code clones, we convert the AST into a Markov chain-based state matrix and use it to achieve scalable code clone analysis. The generation of Markov chains relies on the assumption that the probability of the current state transitioning to the next state depends only on the state before it. Such an assumption drastically diminishes the sophistication of the model, and as a result Markov chain is widely used in many models, such as recurrent neural networks [45] and hidden markov models [8].

To transform an AST into its corresponding state matrix, we need to first define its states within the AST. As shown in Figure 2, we find that there are two categories of nodes in the AST, one being non-leaf nodes and the other being leaf nodes. The non-leaf nodes represent different code syntax types in a method while the leaf nodes of an AST are the source code tokens in a method. For the non-leaf nodes in ASTs, the number is much smaller than the number of tokens since they represent different code syntax types in a method. To achieve a determinate result, we choose the whole dataset of BigCloneBench (BCB) [2] which consists of 250M lines of code as our analysis data. Specifically, we extract the ASTs of all methods and analyze the code syntax types from these trees. Through the analysis report, we obtain a total of 57 code syntax types. For the leaf nodes (*i.e.,* tokens) in ASTs, the exact number is unclear. To make these tokens correspond to a fixed number of states, we consider replacing them with their token types. For example, the token "long" can be replaced by its type "*BasicType*". Similar to analyzing non-leaf nodes, we also choose the whole dataset of BCB [2] to commence our token types collection. After our statistical analysis, we find that 14 types appear in most of ASTs. In fact, the proportion of these 14 types accounts for more than 99.5% of all nodes. Therefore, we choose these 14 types as the final token types and add a *Null* type to represent other types. In final, we collect a total of 72[2] states which consists of 57 code syntax types and 15 token types. Figure 2 shows the part of AST obtained by parsing the original method in Figure 1, and the red parts are leaf nodes represented by token types.

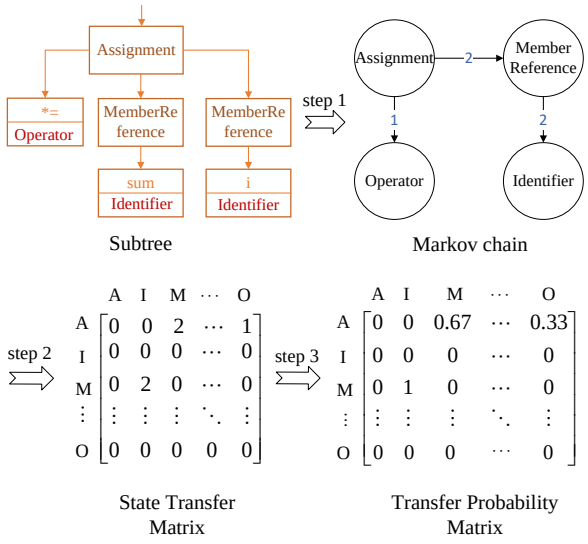

Figure 4: The construction process of transfer probability matrix

After defining 72 states in an AST, we build a Markov chain model to transform the AST into a transfer probability matrix using the three steps indicated in Figure 4. More specifically, relationships between parent and child nodes are considered as state transitions in a Markov chain. By counting the information of two nodes connected by an edge in an AST, the probability of transferring one state to another can be derived. For example, the first part in Figure 4 is a subtree in Figure 2(1), and the subtree has five edges, that is, five sets of state transitions. They can be represented as a Markov chain as shown in the second part of Figure 4. For example, the state *MemberReference* is transferred to the state *Identifier* twice. We use A for *Assignment*, M for *MemberReference*, I for *Identifier*, O for *Operator*, and the resulting state transition matrix is shown in the third part of Figure 4. The value of row A and column M

[2]Due to the limited space, we show the details of these 72 types on our website: https://github.com/CGCL-codes/Amain.

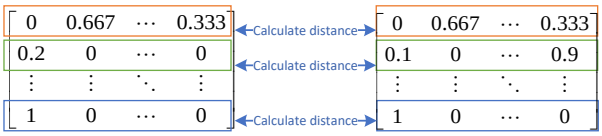

Figure 5: Measure the distance of two matrices to construct the feature vector

is two, which means state *Assignment* has two transfers to state *MemberReference*.

Accordingly, for the whole tree, we define a $72 * 72$ state transfer matrix, where the value of $matrix[i][j]$ indicates the number of times that the child node of the i_{th} state in the AST is the j_{th} state. The value of $matrix[i][j]$ is increased by one whenever the child node of the i_{th} state is the j_{th} state. After obtaining the state transfer matrix, it is transformed into a transfer probability matrix. If the state transfer matrix is M_1 and the transfer probability matrix is M_2, then M_2 is calculated as:

$$M_2\,[i]\,[j] = \frac{M_1\,[i]\,[j]}{\Sigma_{k=0}^{71} M_1\,[i]\,[k]} \tag{1}$$

The transfer probability matrix of the subtree in the first part of Figure 4 is shown in the fourth part of Figure 4. Moreover, since the 15 token types represent non-leafs nodes in ASTs, their out-degree values are all zero. In other words, they have no subsequent state changes. Therefore, the $72 * 72$ state transfer matrix can be changed to a $57 * 72$ matrix to save storage space and runtime overhead.

4.4 Feature Extraction

The purpose of this stage is to collect the distance vectors of two transfer probability matrices. Figure 5 shows the procedure for measuring the distance of the matrices of two methods. For two transfer probability matrices, we compute the distance of their corresponding states (*i.e.,* rows) one by one. The vectors of the first row (*i.e.,* orange part) of the two matrices are taken separately for the distance calculation. After obtaining the distance of the first row (*i.e.,* state), we then take the vectors of the second rows to collect the distance of the second state. Since there are 57 rows altogether, 57 distance values are obtained. These 57 values constitute a 57-dimensional vector, which is called the distance vector of the two transfer probability matrices.

To measure the distance of two vectors, we select four widely used distance computation techniques which are *Cosine distance*, *Euclidean distance*, *Manhattan distance*, and *Chebyshev distance*. These four algorithms have been applied in many areas such as information retrieval, text mining, and data mining [39]. Due to their high effectiveness, we also choose them as our distance computation approaches.

- ***Cosine distance*** evaluates the distance of two vectors by calculating the cosine of the angle between them. The formula is

$$dist_{cos}(A,B) = 1 - cos(\theta) = 1 - \frac{\Sigma_{i=1}^{n} A_i B_i}{\sqrt{\Sigma_{i=1}^{n} A_i^2}\sqrt{\Sigma_{i=1}^{n} B_i^2}} \tag{2}$$

- ***Euclidean distance*** is the true distance between two points in m-dimensional space. The Euclidean distance of a vector is the natural length of the vector, *i.e.,* the distance from the point to the origin. The formula is

$$dist_{euc}(A,B) = \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2} \tag{3}$$

- ***Manhattan distance*** is the sum of the projected distances of a line segment formed by two points in Euclidean space on a fixed rectangular coordinate axis. The formula is

$$dist_{man}(A,B) = \sum_{i=1}^{n} |A_i - B_i| \tag{4}$$

- ***Chebyshev distance*** is a measure in vector space in which the distance between two points is defined as the maximum value of the difference between their coordinates. The formula is

$$dist_{che}(A,B) = \lim_{p\to\infty} \left(\sum_{i=1}^{n} |A_i - B_i|^p \right)^{\frac{1}{p}} \tag{5}$$

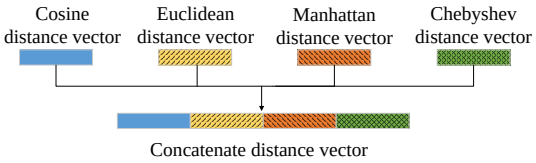

Figure 6: The construction of concatenate distance vector

To achieve more comprehensive experiments, after obtaining four single distance vectors, we concatenate them as a new vector namely *concatenate distance vector*. The construction process is shown in Figure 6. The dimension of four single distance vectors is 57 while is $57 * 4 = 228$ for concatenate distance vector.

4.5 Classification

In our final phase, we focus on training a code clone detector by using machine learning techniques. Specifically, we select several commonly used machine learning methods for classification (*i.e., k-nearest neighbor* (KNN), *random forest* (RF), and *decision tree* (DT)). After feature extraction, we can obtain the distance vectors for all code pairs in the training set. The distance vectors and labels (*i.e.,* one for clone pairs and zero for non-clone pairs) are fed into a machine learning classifier for training, and the resulting model is saved. Two methods to be detected are processed in the three phases described above, and their distance vector is then fed into the model to obtain an output of zero or one. One and zero indicate that they are clone and non-clone, respectively.

5 EXPERIMENTS

In this section, we mainly answer the following research questions:

- *RQ1: What is the detection effectiveness of Amain with different parameters?*
- *RQ2: Can Amain outperform other code clone detectors?*
- *RQ3: What is the runtime overhead of Amain on detecting code clones?*
- *RQ4: Why can Amain detect semantic code clones?*

5.1 Experimental Dataset

We perform evaluations of our method on two datasets: *Google Code Jam* (GCJ) [1] and *BigCloneBench* (BCB) [2], where the GCJ dataset

is the same as the one used by [49]. The programs in this dataset are from the online programming competition held by Google. The programs in each competition problem are written by various programmers, and encompass 1,669 projects from 12 separate competition problems. Projects solving different problems are not analogous, while projects solving the same competing problem are inherently semantically similar and almost syntactically different because they come from distinct programmers. Therefore, we assume that the code pairs in the identical problem are semantic clones (*i.e.,* Type-4 clones) of each other, while the code pairs from different problems are non-clones. In this way, we can obtain 275,570 semantic clone pairs and 1,116,376 non-clone pairs. To make our dataset more balanced, we randomly select 270,000 pairs from 1,116,376 non-clone pairs to commence our experiments.

Table 1: The numbers and the proportions of our used datasets

Type		BCB		GCJ
		Number	Proportion	
Clone Pairs	T1	48,116	17.82%	-
	T2	4,234	1.57%	-
	ST3	21,395	7.92%	-
	MT3	86,341	31.98%	-
	WT3/T4	109,914	40.71%	275,570
	Total	270,000	100.00%	275,570
Non-clone Pairs		270,000	100.00%	270,000

For the second dataset, BCB [2], which consists of more than eight million labeled clone pairs from 25,000 projects. Due to the unclear boundary between Type-3 and Type-4, these two clone types are further divided into three subcategories by a similarity score measured by line-level and token-level code normalizations, as follows: i) *Strongly Type-3* (ST3), where the similarity is between 70-100%, ii) *Moderately Type-3* (MT3), where the similarity is between 50-70%, and iii) *Weakly Type-3/Type-4* (WT3/T4), where the similarity is between 0-50%. Since the number of non-clone code pairs in BCB is 270,000, we also randomly select a total of 270,000 clone pairs from the eight million clone pairs to perform our training and testing phase. The clone pairs include 48,116 T1 pairs, 4,234 T2 pairs, 21,395 ST3 pairs, 86,341 MT3, and 109,914 WT3/T4 pairs. The descriptions of our datasets are shown in Table 1.

5.2 Experimental Settings

Since file-level and program-level code clone detection are too coarse to detect most clones and line-level may detect numerous meaningless clone pairs, we choose a method as our processing granularity because it realizes a specific functionality that fits our need to detect functionally similar clones. Because the programming language of our experimental datasets is Java, we use a Python library namely *Javalang* [4] to parse the Java method to extract ASTs. Moreover, we apply another Python library namely *Sklearn* [3] to implement KNN, RF, and DT classification algorithms.

To ensure the comprehensiveness of our evaluations, we select representative work from each of the code intermediate representations for comparative experiments. Specifically, we compare *Amain* with two token-based methods (*i.e., SourcererCC* [34] and *RtvNN* [44]), four tree-based methods (*i.e., Deckard* [17], *ASTNN* [48], *TBCNN* [26], and *CDLH* [43]), and three graph-based methods (*i.e., SCDetector* [46], *DeepSim* [49], and *FCCA* [14]).

- ***SourcererCC*** [34] : an advanced traditional token-based clone detection tool.
- ***RtvNN*** [44] : an advanced token-based clone detection tool by using a recurrent neural network.
- ***Deckard*** [17] : an advanced traditional AST-based clone detection tool.
- ***ASTNN*** [48] : an advanced AST-based clone detection tool by using a gate recurrent unit network.
- ***TBCNN*** [26] : an advanced AST-based clone detection tool by using a convolutional neural network.
- ***CDLH*** [43] : an advanced AST-based clone detection tool by using a long short-term memory network.
- ***FCCA*** [14] : an advanced graph-based clone detection tool by using hybrid code representations.
- ***DeepSim*** [49] : an advanced graph-based clone detection tool by using a deep neural network.
- ***SCDetector*** [46] : an advanced graph-based clone detection tool by using a Siamese network.

For the selection of parameters of these tools, we choose the parameters they claim to perform best in their papers.[3] We run all experiments on a server with 8 cores of CPU and a GTX 1080 GPU. We use ten-fold cross-validation for training and testing on the dataset, where all the data are divided into ten parts, each part serving as the testing set, and the rest as the training set. The F1, precision, and recall of each testing phase are recorded. Precision is defined as $P = TP/(TP+FP)$. Recall is defined as $R = TP/(TP+FN)$. F1 is defined as $F1 = 2*P*R/(P+R)$. Among them, *true positive* (TP) represents the number of samples correctly classified as clone pairs, *false positive* (FP) represents the number of samples incorrectly classified as clone pairs, and *false negative* (FN) represents the number of samples incorrectly classified as non-clone pairs.

5.3 RQ1: Comparison of Different Methods

To measure the performance of various distance calculation methods and machine learning algorithms, we implement experiments on the GCJ dataset and the BCB dataset. As aforementioned, we choose four different distance calculation methods (*i.e., Cosine distance*, *Euclidean distance*, *Manhattan distance*, and *Chebyshev distance*) and construct another new distance vector (*i.e.,* concatenate distance vector) to commence our experiments. After obtaining all feature vectors, we select different machine learning algorithms (*i.e.,* KNN, RF, and DT) for training and testing. For K in KNN, we choose one and three since they are the most widely used. For RF, we experiment with different numbers of depth parameters and find that 64 is a better parameter. For DT, we use the default parameters in the *Sklearn* library.

Through the results in Figure 7, we can derive two conclusions. One is that *Amain* can maintain the best detection performance when we use concatenate distance vectors to train an RF machine learning classifier. RF algorithm is essentially an improvement on the DT algorithm by combining several decision trees. The classification power of a single tree may be negligible, but after randomly

[3]Due to the limited space, the details of these parameters can be found on our website: https://github.com/CGCL-codes/Amain.

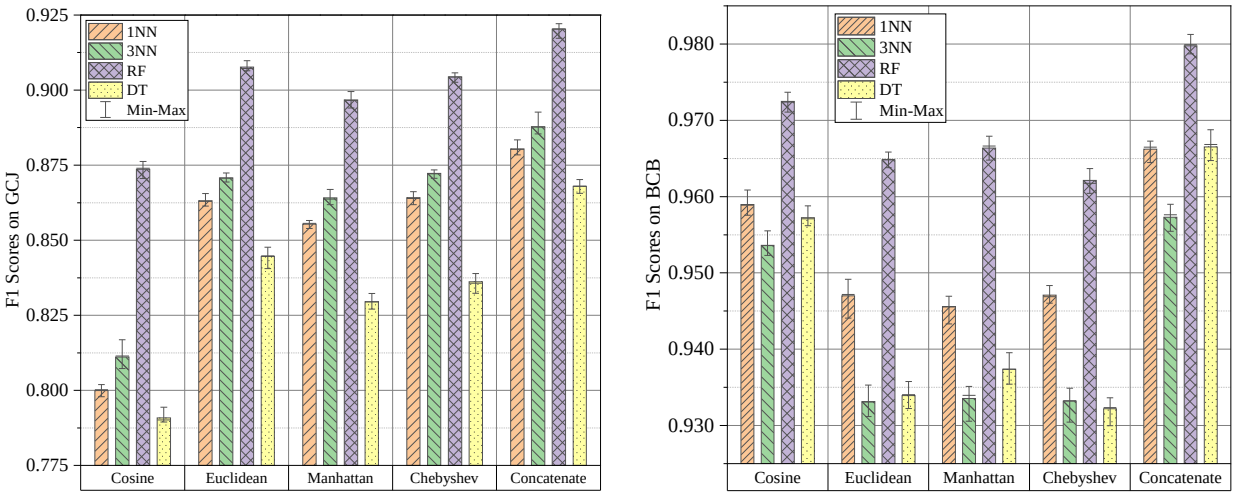

Figure 7: F1 score on GCJ and BCB datasets

generating a multitude number of decision trees, a test sample can statistically select the most probable classification from the classification results of each tree. Therefore, the experimental results using RF algorithm are better. The differences between the results of the five distance methods are slight, but the results of concatenate distance are marginally superior to the others. This is because four single distance calculation methods measure the distance of two vectors from only one perspective. However, a concatenate distance vector is constructed from these four single distance vectors, which may contain more comprehensive details of two vectors.

Another conclusion is that the ten F1 scores of ten-fold cross-validations do not differ much. For example, the maximum and minimum values of the RF algorithm with the concatenate distance differ by only 0.0025 on the BCB dataset and by only 0.0048 on the GCJ dataset. Thus, the average value can give a good indication of the detection effectiveness of each distance calculation method and machine learning algorithm. Therefore, in the overall effectiveness experiments in the next subsection, we only use the average value of the metrics to represent the detection effectiveness.

5.4 RQ2: Overall Effectiveness

From the previous subsection, it is evident that the best detection results are obtained by using the concatenate distance calculation method in the feature extraction phase and RF machine learning algorithm in the classification phase. Therefore, we use them for the overall effectiveness experiments.

5.4.1 Results on Google Code Jam. First, we perform evaluations on the GCJ dataset. As we mentioned previously, code pairs from the identical competing problem on the GCJ dataset are inherently semantically similar and are nearly impossible to be syntactically similar. Therefore, we regard all similar code pairs as semantic clones (*i.e.,* Type-4 clones) and run experiments on these pairs to evaluate the effectiveness of *Amain* in detecting semantic clones. The evaluation results of *SourcererCC* [34], *RtvNN* [44], *Deckard* [17], *ASTNN* [48], *TBCNN* [26], *CDLH* [43], *SCDetector* [46], *DeepSim* [49], *FCCA* [14], and *Amain* are presented in Table 2.

Table 2: Results of clone detection on GCJ and BCB datasets

Group	Method	GCJ			BCB		
		R	P	F1	R	P	F1
Token-based	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-based	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-based	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
Our method	Amain	**0.91**	0.93	**0.92**	0.97	**0.99**	**0.98**

Token-based approaches: *SourcererCC* has both poor recall and precision. This is because *SourcererCC* only takes into account the overlapping similarity of tokens between two methods. Given two methods $M1$ and $M2$, the overlapping similarity $S(M1, M2)$ is the ratio of the number of identical tokens shared by $M1$ and $M2$ to the number of larger tokens in $M1$ and $M2$. Due to the lack of consideration of program semantics, *SourcererCC* cannot handle semantic clones and thus has low recall and precision. *RtvNN* has high recall but low precision. This means that *RtvNN* can detect almost all code pairs as clones. This is because *RtvNN* relies only on a simple distance metric to measure the similarity of a pair of code pairs. As described in [49], the distance between most methods is in the range of [2.0, 2.8] using *RtvNN* for computation. The precision of *RtvNN* can be increased to 90% by lowering the distance threshold, it will also bring a rapid decline in the recall to below 10%. Hence, it has a low F1 score.

Tree-based approaches: The detection performance of *Deckard* and *CDLH* is not so good. The reason is that *Deckard* detects clones by clustering the eigenvectors of each subtree using predefined rules for both functions. However, more than half of the code pairs do not have the same tree structure, resulting in a low recall and precision of detection in the dataset. *CDLH* uses an AST-based long

short-term memory network to learn the representation of hash functions, structural information, and code fragments. As it only considers lexical and syntactic code features, it does not have good detection results. The other two tree-based methods *ASTNN* and *TBCNN* have relatively good ability. This is because *ASTNN* uses a bottom to top (*i.e.,* leaf node to root node) aggregation for AST to detect clones. Thus it is better than the same AST-based method *Deckard* which only clusters the features at the root of the tree. However, the operation of segmentation of the AST by *ASTNN* may result in some semantics loss, which leads to a marginally lower recall. *TBCNN* captures the structural features of the AST by sliding convolutional kernels and therefore has good detection results. However, the convolutional layer has difficulty in capturing long-range contextual information if the AST is deep or has many nodes. Also, the operation of treating the AST as a binary tree aggravates the problem of long-term dependency on the original semantics of the source code. Therefore, the detection result of *TBCNN* is not the best.

Graph-based approaches: For *DeepSim*, it abstracts the variables and basic blocks of code and the relationships between them into a binary matrix, which is then fed into a deep learning model to detect clones. Since it considers the semantics of a method, it can achieve ideal performance on semantic clone detection. For *SCDetector*, it assigns semantic details to tokens by analyzing the centrality of each basic block in the CFG, thus it is dependent on the common tokens between two methods, leading to several false negatives when the same functionality is implemented using different APIs and different graph structures. *FCCA* extracts different representations of code, including unstructured representations (*i.e.,* tokens) and structured representations (*i.e.,* AST and CFG), and feeds the them into a deep learning model with an attention mechanism to detect clones. The comprehensive hybrid code representation enables *FCCA* to detect most semantic code clones.

In summary, *SourcererCC*, *Deckard*, and *RtvNN* do not have the ability to handle semantic clones. Meanwhile, compared to *CDLH*, *ASTNN*, *TBCNN*, *SCDetector*, *DeepSim*, and *FCCA*, *Amain* can handle more semantic code clones on GCJ dataset.

Table 3: *Amain*'s Recall, Precision, and F1 in detecting each type of clone on BCB

Metrics	T1	T2	ST3	MT3	WT3/T4
Recall	1.00	1.00	1.00	0.99	0.92
Precision	1.00	1.00	1.00	0.99	0.98
F1	1.00	1.00	1.00	0.99	0.95

5.4.2 Results on BigCloneBench. We then compare *Amain* with our comparative tools on the BCB dataset. The detection results are presented in Table 2. From the table, we can see that *Amain* outperforms all other detectors in terms of precision and F1 score. Such results suggest that *Amain* is extremely well equipped to detect code clones. Moreover, we also observe that most clone detection tools have better detection results on BCB than GCJ. It is reasonable because the clone pairs in BCB are deliberately constructed by experts. Many clone pairs share a similar code structure, with only a few differences such as the order of API calls. However, the programs in the GCJ dataset are all implemented by different programmers, resulting in more complex and different code structures. For *SourcererCC*, *Deckard*, and *RtvNN*, they both have low recall and high precision. This is because these tools detect two code segments as clones only when they are extremely similar. Therefore, they are only able to detect syntactic code clones on BCB, but not semantic clones.

Next, we analyze the recall, precision, and F1 for each type and compare them with other code clone detection techniques. Table 3 shows the corresponding results. It can be seen that *Amain* can achieve more than 99% scores for all three metrics in detecting T1, T2, ST3, and MT3. When detecting WT3/T4, the F1 reaches a score of 95%, the precision reaches 98%, and the recall reaches 92%. Such results indicate that *Amain* has the ability to detect semantic code clones.

Table 4: F1 for each clone type on BCB

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
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
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
Our method	Amain	**1.00**	**1.00**	**1.00**	**0.99**	**0.95**

Table 4 shows the evaluation results of *Amain* and other comparative tools in detecting five types of code clones. From the results, we see that *Amain* outperforms all other clone detectors in detecting all types of code clones. Especially when detecting WT3/T4, the F1 scores of *SourcererCC*, *Deckard*, *RtvNN*, *ASTNN*, *TBCNN*, and *CDLH* are 2%, 2%, 0%, 92%, 86%, and 82%, respectively, while *Amain* can maintain an F1 score of 95%. Such results demonstrate the superiority of *Amain* in detecting WT3/T4 code clones. For *SCDetector*, *DeepSim*, *FCCA*, and *ASTNN*, they can also achieve ideal performance in detecting WT3/T4 code clones. However, they all use deep neural networks to train classifiers, meaning that they require GPUs to accomplish their training phases. For *Amain*, we apply simple machine learning models to train our classifier. Therefore, CPUs are enough for us. In other words, *Amain* requires less computational resources than *SCDetector*, *DeepSim*, *FCCA*, and *ASTNN*.

5.5 RQ3: Scalability

In this part, we focus on evaluating the scalability of *Amain* and our nine comparative systems. Specifically, we first randomly select one million code pairs from the GCJ dataset as the analysis targets. Then we run *Amain* and comparative tools on these pairs ten times and record their runtime overhead. From the results in our previous subsection, we know that *Amain* can maintain the best detection performance when extracting concatenate distance vectors and using RF to train a classifier. So we record the training and testing runtime overheads when using concatenate distance vectors to train an RF model. In particular, our training overhead includes the runtime of data preprocessing, analysis, and model training. Table 5 describes the average runtime overhead of ten runs of each tool and the fluctuation of the overhead caused by the impact of

the physical environment (*e.g.,* CPU usage). Through the results in Table 5, it can be seen that even if we run the same tool on the same machine, the cost of each run can be different. This is mainly because the usage status of the same machine at different times may be different. Overall, this inaccuracy is around 7%. To mitigate this issue, we run each tool ten times and choose its average as the final runtime overhead.

Table 5: Time performance on analyzing one million code pairs

Group	Method	Training	Prediction
Token-based	SourcererCC	-	16s±1s
	RtvNN	5,206s±364s	35s±2s
Graph-based	SCDetector	2,937s±205s	139s±9s
	DeepSim	13,545s±948s	34s±2s
	FCCA	56,769s±3,973s	91s±6s
Tree-based	Deckard	-	72s±4s
	ASTNN	16,096s±1,126s	2,894s±202s
	TBCNN	41,168s±2,881s	86s±6s
	CDLH	45,317s±3,172s	90s±6
Our method	Amain	1,017s±71s	18s±1s

For *SourcererCC* and *Deckard*, they do not have training phases, thus their training runtime are both zero. For other tools (*i.e., RtvNN*, *SCDetector*, *DeepSim*, *FCCA*, *ASTNN*, *TBCNN*, and *CDLH*), they are all deep learning-based approaches which require GPUs to assist their training phases. As shown in Table 5, we can see that even if they use GPUs for training and testing, their average runtime overheads are higher than that of *Amain*, which only uses CPUs for training and testing. Such results indicate that *Amain* is more scalable than *RtvNN*, *SCDetector*, *DeepSim*, *FCCA*, *ASTNN*, *TBCNN*, and *CDLH*. For another recent state-of-the-art AST-based code clone detector (*i.e., ASTNN*), results in Table 5 show that *ASTNN* takes an average of 16,096 seconds to train and 2,894 seconds to test, whereas *Amain* takes only 1,035 seconds (*i.e.,* 1,017 seconds for training and 18 seconds for testing) to complete code clone detection of one million code pairs. In other words, the training runtime of *Amain* is 15.8 (*i.e.,* $16,096/1,017 = 15.8$) times faster than that of *ASTNN*. Similarly, *Amain*'s prediction runtime is also shorter than *ASTNN*'s. When given a trained model, the runtime overhead of *ASTNN* to detect clones is 2,894 seconds, while *Amain* only requires 18 seconds. Overall, it can be said that *Amain* is 18.3 (*i.e.,* $(16,096+2,894)/(1,017+18) = 18.3$) times faster than *ASTNN*.

In summary, *Amain* is not as fast as *SourcererCC* due to the consideration of tree details. However, because of the use of a machine learning algorithm and a Markov chains model to convert ASTs into matrices, *Amain* is more scalable than *RtvNN*, *SCDetector*, *DeepSim*, *FCCA*, *ASTNN*, *TBCNN*, and *CDLH*.

5.6 RQ4: Interpretability

To illustrate how *Amain* detects semantic clones, we first train an RF model on GCJ dataset. Since RF algorithm is interpretable, we collect the importance of all features used in *Amain*. After ranking these 228 features by their weights computed by our RF technique, we figure out which features are more important in detecting semantic clones. Due to the limited space, we only show the top 20 features in Table 6. Observing the table, we find that four features recur in the top 20 features: *StatementExpression*, *ArrayCreator*, *ArraySelector*, and *BlockStatement*. Such results suggest that these four types of nodes in ASTs are the most important for distinguishing whether code pairs are semantic clones or not.

Table 6: Top 20 features of *Amain* in detecting semantic code clones

Rank	Feature Name	Weight
1	StatementExpression_cosine	0.029142
2	ArrayCreator_euclidean	0.026448
3	ArraySelector_euclidean	0.025292
4	BlockStatement_manhattan	0.024604
5	ArrayCreator_chebyshev	0.022709
6	ArraySelector_chebyshev	0.021914
7	ArraySelector_manhattan	0.021804
8	ArrayCreator_manhattan	0.021659
9	ArrayCreator_cosine	0.021011
10	StatementExpression_chebyshev	0.020044
11	StatementExpression_manhattan	0.019399
12	StatementExpression_euclidean	0.019094
13	BlockStatement_euclidean	0.018799
14	BlockStatement_cosine	0.017125
15	VariableDeclarator_cosine	0.016667
16	MemberReference_cosine	0.015774
17	BinaryOperation_cosine	0.015754
18	ArraySelector_cosine	0.015429
19	BinaryOperation_manhattan	0.014952
20	BlockStatement_chebyshev	0.014904

For *StatementExpression*, it is always connected to the type of statement. For example, code "`int i=1`" of the original code in Figure 1 is an assignment statement, so in the corresponding subtree in Figure 2, the *StatementExpression* node connects to the *Assignment* node. Another node type that is often connected behind the *StatementExpression* is *MethodInvocation*, which indicates the invocation of a method. Method invocations are often strongly associated with program semantics. Therefore, different node types connected by *StatementExpression* can reflect different semantic information of the code and thus have high importance.

ArrayCreator represents a declaration of an array, such as "`new int[3]`". *ArraySelector* represents a selection of array elements, such as "`a[index]`". *ArrayCreator* connects nodes that reflect the type and size of an array, and *ArraySelector* connects nodes that reflect the selection of array elements. Both array-related node types reflect the details about array, and implementing the same functionality may require a similar data structure. Therefore, the relevant information of the array is also important to maintain code semantics.

BlockStatement is usually after *SwitchStatementCase*, *ForStatement*, *IfStatement*, *WhileStatement*, *MethodDeclaration*, and other node types that can control whether the following statements are executed or not. The edges containing the *BlockStatement* node are likely to reflect the control information of the program, and the control flow is a reflection of the program semantics. The *BlockStatement* node type is followed by various statement types, such as the most common *StatementExpression*, or *ReturnStatement*, *IfStatement*, *ContinueStatement*, *LocalVariableDeclaration*, etc. The node type following *BlockStatement* abstracts the statements contained

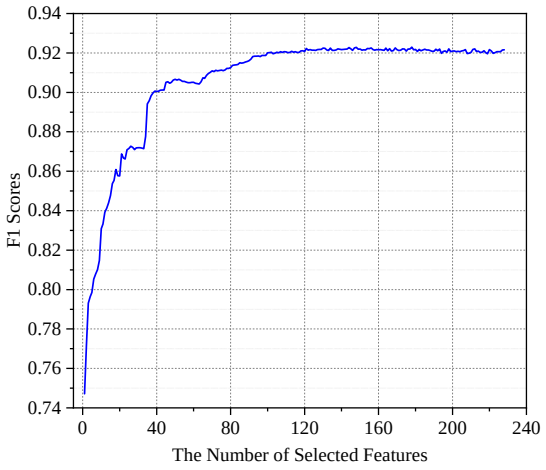

Figure 8: F1 scores of *Amain* when selecting different numbers of features

within a statement block. Different statement types represent different program details, and different program details imply different semantics. It is probably for this reason that the *BlockStatement* node type has high importance.

To obtain a more intuitive view of how these features work, we sort each element of the concatenate distance vector by the importance of its corresponding feature. Then we take the top n (n from 1 to 228) features in turn and record the F1 scores for different lengths of the vector using a ten-fold cross-validation method. Through the results in Figure 8, we find that an F1 score of 0.7461 can be achieved using only one feature (*i.e., StatementExpression_cosine*) for classification. Besides, with the continuous increase of important features, the detection performance becomes increasingly effective. As the number of features gradually increases from 1 to 50, the improvement of F1 score is very significant, and the increase of F1 score is relatively slow from 50 to 100. When the number of features increases to around 100, *Amain* can almost achieve the best results, and the F1 is 0.92. This indicates that the top 100 important features have been able to detect semantic clones well. In the future, we plan to use different feature selection algorithms to find the most suitable feature combination for better detection performance.

6 DISCUSSION

6.1 Threats to Validity

Code pairs in the BCB dataset are labeled by several experts, and not all data belong to semantic code clones. Therefore, BCB may not be representative of the entire open-source projects. To mitigate the threat, we choose another open-source dataset which consists of 1,669 projects from 12 separate competition problems held by Google. To obtain fixed-length states from ASTs, we select a total of 15 token types to commence our experiments. The selection of these types may cause some inaccuracies since the total number of token types parsed by *Javalang* is not clear. To mitigate the situation, we perform statistical analysis on the whole BCB dataset to select the types with a high number of occurrences and add a *Null* type to represent the remaining types. The calculation of runtime overheads of *Amain* and its comparative tools may also cause some inaccuracies due to the different machine statuses such as CPU usage. We mitigate the threat by conducting evaluations ten times and reporting the average runtime overhead in our paper.

6.2 Discussion

6.2.1 Why does Amain perform better? First, *Amain* maintains the program details by generating the AST of a method, which enables it to detect Type-4 with a high degree of precision. Second, the use of Markov chains in *Amain* converts the complex ASTs into matrices that are convenient to store and manipulate, while preserving the tree details of the code and eliminating sophisticated tree matching. Third, compared to *SourcererCC* [34] which computes the similarity directly after parsing the program, *Amain* represents the similarity of two codes by measuring the distance between two matrices, and then puts the distance vector into a machine learning classifier for training and testing, which significantly improves the efficiency and accuracy of *Amain*.

6.2.2 Programming language generalizability. In this paper, we mainly focus on Java code clone detection. In reality, our approach can extend to other programming languages with small modifications. For instance, we can leverage *pycparser* [5] as the lexical analysis tool to extract the AST of C source code, then the same Markov chains model building and machine learning classifier training can be adopted to detect C code clones.

6.2.3 Future work. In our paper, we select four distance calculation methods to measure the distance between two matrices and four machine learning algorithms to train code clone classifiers. Experimental results show that the use of different distance calculation methods and different machine learning algorithms can achieve different detection effectiveness. In our future work, we plan to select more distance calculation methods and machine learning algorithms to find a more suitable combination for achieving better detection performance. Since most code clone detection tools are not open source, we only select nine comparative systems. We will conduct a detailed comparative analysis on more systems in the future. From results in Figure 8, we observe that the top 100 important features may be enough for *Amain* to detect semantic code clones. In our future work, we would like to apply different feature selection techniques to figure out the most suitable combination of these features. In this way, *Amain* may achieve more scalable and effective semantic code clone detection.

7 RELATED WORK

In this section, we concentrate on the current research associated with clone code detection. The existing clone code detection methods are mainly classified into text-based, token-based, tree-based, graph-based, and metric-based tools.

For text-based clone detection techniques [10, 16, 19, 29, 32, 47], these techniques convert no or little source code and compare two code segments by treating the source code as a series of lines or strings. Johnson [19] detects clones by fingerprint matching method. [10] detects clones by treating lines of code as strings and then using string matching methods for similarity estimation. Therefore, this method can only find extremely similar code clones and does not have the ability to detect semantic clones. [32] detects potential clones by matching the code text using an algorithm of the longest

common subsequence. These methods do not consider the logic of the code fragment and the semantics of the program, but only consider the code as a simple string and perform the detection of code clones by string matching. Therefore, only clones with high text similarity can be detected, and it is almost impossible to find Type-3 and Type-4 code clones.

For token-based clone detection techniques [11, 12, 15, 20, 23, 34, 42], these methods convert the target source code into a token sequence by lexical analysis of the program code, and then detect code clones by scanning the token sequences and finding duplicate token subsequences. [11] creates a map by comparing additions and deletions between two clones, and then detects clones. *CCFinder* [20] first extracts the token sequence from the program code and then develops transformation rules to convert the token sequence, which can be used to detect Type-1 and Type-2 clones. *CClearner* [23] is the first fully token-based clone detector that extracts tokens from clone and non-clone pairs to train a classifier, which is then used to detect clones in the codebase. *SourcererCC* [34] detects clones by comparing marked overlaps, which is good at detecting format conversions and renaming, and it is well scalable. *CCAligner* [42] is the best tool to detect the performance of large gap clones. It is capable of detecting Type-1, Type-2, and Type-3 clones by performing similarity calculations using novel electron mismatch indices and asymmetric similarity coefficients. However, the token-based approach cannot handle Type-4 clones. *CPPCD* [15] is a CP-based potential clone detector by generating *Clone Probability* (CP) values and CP distribution maps for developers to determine whether two methods are a clone.

For tree-based clone detection techniques [9, 17, 18, 24, 28, 43, 48], these methods parse the program into a parse tree or abstract syntax tree. The clone is then detected by tree matching. *Deckard* [17] uses *Locality Sensitive Hashing* (LSH) to detect clones by clustering the similar vectors obtained by computing AST for any language with grammatical regulations. *CDLH* [43] first normalizes the abstract syntax tree to transform it into a binary tree, and then encodes the representation of the tree to form vectors using Tree-LSTM [38]. *ASTNN* [48] divides the abstract syntax tree into several subtrees by predefined rules, then encodes each subtree separately to form vectors, and integrates the subtree vectors into a final vector representation using a bidirectional RNN model. [24] is the first method that uses AST-path for clone detection. It represents each code fragment by a set of AST-paths, and then puts these paths into a compare-aggregate model to learn a vector representation, and detects whether a pair of codes is cloned by detecting the similarity of a pair of vectors.

For graph-based clone detection techniques [21, 22, 41, 46, 49, 50], programs are represented as different graph representations, such as *Program Dependency Graph* (PDG) and *Control Flow Graph* (CFG). Most techniques detect clones by subgraph matching, e.g. [22] and [21] use subgraph matching to identify similar code fragments. However, the subgraph matching approach usually has a large time overhead, so *CCSharp* [41] proposes two methods to reduce the overhead, which is a modification of the graph structure and filtering of the feature vectors. [49] converts the code similarity problem into a binary classification problem to deal with code cloning. It converts the code control flow and data flow in the graph into semantic high-dimensional sparse matrices, and then puts the transformed binary feature vector into a deep learning model to measure the functional similarity of the codes. It can efficiently detect code clones that are syntactically extremely different but functionally similar. *SCDetector* [46] treats the control flow graph as a social network, extracts centrality for each block to form semantic tokens with the scalability of a token-based approach and the accuracy of a graph-based approach, and it can detect Type-4 clones with extremely high accuracy.

For metric-based clone detection techniques [6, 13, 25, 27, 30, 33, 35, 37, 40], these methods use the properties of the code to measure the similarity of two code fragments. The metrics are sometimes obtained from a tree or graph representation of the source code, and sometimes directly from the source code. For example, both [6] and [25] use AST as a code feature to identify code clones. [27] detects clones with respect to different classes of metrics (*e.g.*, classes, couplings, and hierarchies) obtained directly from the source code.

Text-based and token-based methods are the fastest, but they do not take into account the program semantics and cannot detect semantic code clones (Type-4 clones). Graph-based methods can detect semantic clones, but building graph models of programs generally requires compilation and many clones are small fragments that cannot be compiled successfully. Tree-based approaches are increasingly used because they do not require compilation and can preserve the program details. However, the AST generated by a few lines of code can be very complex, resulting in high analysis overhead and low efficiency. In our paper, we build a Markov chains model to transform the complex tree analysis into simple state transitions. After obtaining the distance vector between all states, we train a classifier by using some traditional machine learning algorithms. The use of Markov chains model and machine learning model enable *Amain* to achieve scalable and effective semantic code clone detection.

8 CONCLUSION

In this paper, we propose *Amain*, a novel AST-based method for detecting semantic code clones. To avoid heavy-weight tree matching, *Amain* transforms the complex AST into simple Markov chains and build a model to achieve efficient code clone scanning. We evaluate *Amain* on two widespread datasets, BigCloneBench [2, 36] and Google Code Jam [1]. Our experiments show that *Amain* outperforms nine state-of-the-art code clone detection systems (*i.e.*, *SourcererCC* [34], *RtvNN* [44], *Deckard* [17], *ASTNN* [48], *TBCNN* [26], *CDLH* [43], *SCDetector* [46], *DeepSim* [49], and *FCCA* [14]). Compared to another current state-of-the-art AST-based code clone detector *ASTNN*, *Amain* is about 18.3 times faster in analyzing code clones.

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 National Science Foundation of China under Grant No. U1936211 and Hubei Province Key R&D Technology Special Innovation Project under Grant No. 2021BAA032.

REFERENCES

[1] 2017. Google Code Jam. https://code.google.com/codejam/past-contests.
[2] 2022. BigCloneBench. https://github.com/clonebench/BigCloneBench.

[3] 2022. An open source machine learning library that supports supervised and unsupervised learning. (scikit-learn). https://scikit-learn.org/stable/.

[4] 2022. A pure Python library for working with Java source code, provies a lexer and parser targeting Java 8. (Javalang). https://pypi.org/project/javalang/.

[5] 2022. pycparser is a complete parser of the C language. https://pypi.python.org/pypi/pycparser/.

[6] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. 1999. Measuring clone based reengineering opportunities. In *Proceedings of the 6th International Software Metrics Symposium (ISMS'99)*. 292–303.

[7] 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.

[8] Yoshua Bengio, Renato De Mori, Giovanni Flammia, and Ralf Kompe. 1992. Global optimization of a neural network-hidden Markov model hybrid. *IEEE Transactions on Neural Networks* 3, 2 (1992), 252–9.

[9] Sergej Chodarev, Emilia Pietrikova, and Jan Kollar. 2015. Haskell clone detection using pattern comparing algorithm. In *Proceedings of the 13th International Conference on Engineering of Modern Electric Systems (EMES'15)*. 1–4.

[10] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings of the 1999 International Conference on Software Maintenance (ICSM'99)*. 109–118.

[11] 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.

[12] 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.

[13] Syed Mohd Fazalul Haque, V. Srikanth, and E. Sreenivasa Reddy. 2016. 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)*. 340–344.

[14] 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.

[15] 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.

[16] Shruti Jadon. 2016. Code clones detection using machine learning technique: support vector machine. In *Proceedings of the 2016 IEEE International Conference on Computing, Communication and Automation (ICCCA'16)*. 299–303.

[17] 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.

[18] 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.

[19] J. Howard Johnson. 1994. Substring matching for clone detection and change tracking. In *Proceedings of the 1994 International Conference on Software Maintenance (ICSM'94)*. 120–126.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] J. F. Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. 1999. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99)*. 49–56.

[28] Jayadeep Pati, Babloo Kumar, Devesh Manjhi, and Kaushal Kumar Shukla. 2017. A comparison among ARIMA, BP-NN, and MOGA-NN for software clone evolution prediction. *IEEE ACCESS* 5, 1 (2017), 11841–11851.

[29] Chaiyong Ragkhitwetsagul and Jens Krinke. 2017. Using compilation/decompilation to enhance clone detection. In *Proceedings of the 11th IEEE International Workshop on Software Clones (IWSC'17)*. 1–7.

[30] Chaiyong Ragkhitwetsagul, Jens Krinke, and Bruno Marnette. 2018. A picture is worth a thousand words: Code clone detection based on image similarity. In *Proceedings of the 12th IEEE International Workshop on Software Clones (IWSC'18)*. 44–50.

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

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

[33] 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.

[34] 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.

[35] M. Sudhamani and Lalitha Rangarajan. 2016. Code clone detection based on order and content of control statements. In *Proceedings of the 2nd IEEE International Conference on Contemporary Computing and Informatics (ICCCI'16)*. 59–64.

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

[37] Jeffrey Svajlenko and Chanchal K. Roy. 2017. Fast and flexible large-scale clone detection with cloneWorks. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE'17)*. 27–30.

[38] Kai Sheng 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).

[39] 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.

[40] Masateru Tsunoda, Yasutaka Kamei, and Atsushi Sawada. 2016. 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)*. 15–16.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] Holger Wigström. 1974. A model of a neural network with recurrent inhibition. *Kybernetik* 16, 2 (1974), 103–12.

[46] 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.

[47] Dongjin Yu, Jie Wang, Qing Wu, Jiazha Yang, Jiaojiao Wang, Wei Yang, and Wei Yan. 2017. Detecting Java code clones with multi-granularities based on bytecode. In *Proceedings of the 41st IEEE Annual Computer Software and Applications Conference (COMPSAC'17)*. 317–326.

[48] 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.

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

[50] 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.