IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

Semantically Enhanced AST Vectorization for Scalable Code Clone Detection

Kakumanu Kailash Nath, Cheemaladinne Lakshmi Siri, Gugulothu Karthik
B.Tech Students, Dept. of Computer Science and Engineering
Dr. R. Lakshmi Tulasi
Professor, Dept. of Computer Science and Engineering
R.V.R & J.C College of Engineering, Guntur, Andhra Pradesh, India

Abstract: *Code clone detection is a critical task in software engineering, aimed at identifying duplicated code segments that can hinder maintainability and increase the risk of defect propagation. Code2Img presents a scalable solution by converting Abstract Syntax Trees (ASTs) into image-based vector representations, enabling effective detection of syntactically complex clones. However, its reliance on structural similarity limits its ability to capture deeper semantic relationships. In this work, we enhance Code2Img by integrating a semantic transition scoring mechanism. We compute heuristic scores for AST node transitions based on node depth and connectivity within local function contexts, reflecting their semantic roles. These scores are further weighted using Inverse Document Frequency (IDF) across a broader corpus to emphasize informative yet uncommon transition patterns. The enriched semantic scores are incorporated into the vector representation used by Code2Img, enhancing its capacity to detect nuanced Type-3 clones and extending potential coverage toward Type-4 clones. Empirical evaluation demonstrates that our approach preserves the scalability of the original framework while significantly improving semantic clone detection performance.*

Keywords: *Code clone*

I. INTRODUCTION

1.1. Background and Motivation

Code reuse, often implemented through copying, pasting, and subsequent modifications, is a prevalent strategy in contemporary software development practices. This approach offers developers a seemingly convenient method for swiftly implementing new functionalities, leveraging existing codebases and reducing the necessity for de novo code creation. However, this convenience brings with it inherent risks that warrant careful consideration. The introduction of security vulnerabilities and the amplification of maintenance costs are primary concerns associated with the widespread adoption of code cloning methodologies [1], [5]. Cloned code segments may inadvertently propagate existing vulnerabilities present in the original source, thereby creating multiple points of weakness within the encompassing system. Furthermore, the maintenance of cloned code can pose significant challenges, necessitating the application of updates and bug fixes to numerous instances of similar code, which in turn elevates the probability of introducing inconsistencies and augmenting the overall maintenance burden.

Clone detection has emerged as a critical area within software engineering, aimed at mitigating the aforementioned risks through the identification of similar code segments [5], [15], [19]. By systematically identifying code clones, developers can gain enhanced insights into code dependencies, detect potential security vulnerabilities that may proliferate across cloned segments, and strategically allocate maintenance efforts to areas exhibiting the highest degree of code duplication. Effective clone detection methodologies facilitate improved code management practices, reduce redundancy inherent in cloned code segments, and ultimately contribute to the enhancement of both the quality and security attributes of software systems [7], [15]. Moreover, the application of clone detection techniques can extend to the identification of violations pertaining to software licenses, thereby ensuring compliance with established licensing

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

agreements and precluding potential legal ramifications stemming from the unauthorized replication or utilization of code [5], [43], [44].

Scalable and complicated clone detection represent the two core demands in the field of clone detection. Scalable clone detection specifically addresses the imperative to efficiently analyze large codebases, a prerequisite for the practical application of clone detection methodologies in real-world software projects [15], [16]. The capacity to process millions of lines of code within reasonable timeframes and with judicious resource utilization constitutes a fundamental requirement for scalable clone detection systems [14], [2]. Complicated clone detection, conversely, centers on the identification of clones that transcend mere identical copies, encompassing instances where code segments have undergone substantive modifications, thereby rendering their detection more arduous [7], [8], [11]. Such modifications may encompass statement reordering, insertions, deletions, and variable name changes [23]. Addressing both scalability and the detection of complicated clones is essential for developing effective tools that can support the development and maintenance of modern software systems [53].

1.2. Limitations of Existing Approaches

Existing clone detection approaches can be broadly categorized into text-based, token-based, graph-based, and tree-based methods, each characterized by its own distinct strengths and limitations. Text-based and token-based methods are generally efficient for scalable clone detection, making them suitable for analyzing large codebases [5], [15], [16]. These methodologies treat source code as either plain text or token sequences, proceeding to calculate similarities directly from these representations. However, they often struggle with complicated clone types because they lack consideration for the underlying code structure [7], [15]. Text-based methods exhibit sensitivity to minor code variations, encompassing whitespace alterations and commentary, while token-based methods are susceptible to perturbations in variable names and statement ordering [5], [6].

Methods based on intermediate representations of code, such as graph-based and tree-based approaches, can effectively detect complex clones by capturing the structural and semantic information of code [8], [9], [10], [11]. Graph-based approaches employ program dependency graphs (PDG) or control flow graphs (CFG) to represent code structure, thus enabling the identification of clones even in the presence of substantial modifications [8], [11]. Tree-based methods, on the other hand, leverage abstract syntax trees (AST) to capture the hierarchical structure of code, facilitating the detection of syntactically similar yet non-identical clones [10], [19], [20]. However, graph-based methods are limited by long generation times and complex graph structures, leading to high runtime overhead [9], [11]. The computational complexity inherent in constructing and comparing intricate graphs can render graph-based methodologies impractical for large-scale clone detection endeavors [2], [9].

Tree-based methods, while offering improvements in speed relative to graph-based approaches, nonetheless encounter scalability challenges stemming from the inherent complexity of ASTs [10], [20], [53]. The size and complexity of ASTs can lead to high memory consumption and long detection times, limiting their scalability for very large codebases [19], [53]. Abstract Syntax Tree (AST)-based methods, while capable of capturing code structure, face challenges in large-scale clone detection because the complex structure of ASTs leads to high memory consumption and long detection times [19], [20]. The computational cost of comparing ASTs directly can be significant, especially when dealing with large and complex codebases. The main limitation of the AST-based method is the tree's complex structure, so it has to take up a large amount of memory and a long time for the trees matching. This makes it difficult to meet the needs of scalable clone detection [53].

1.3. Semantically Enhanced AST Vectorization Approach and Contributions

The "Semantically Enhanced AST Vectorization" approach addresses the challenge of achieving both effectiveness and scalability in detecting complicated clones from large-scale source code [53]. It achieves this by transforming Abstract Syntax Trees (ASTs) into vector representations that capture both syntactic and semantic information. This approach leverages the strengths of AST-based methods [10], [19], [20] while mitigating their limitations by using vector representations that are more compact and efficient to compare [53].

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

The approach involves several key steps. First, ASTs are generated from the source code to capture the syntactic structure [24]. Second, semantic information is extracted from the ASTs using node metrics such as frequency, degree, and depth of a node [10], [11]. Third, the syntactic and semantic information is combined to create a vector representation of each code fragment [53]. Finally, clones are detected by calculating the similarity between the vector representations, allowing for efficient and scalable clone detection [2], [15].

The key contribution of this approach is a novel method to efficiently represent code fragments as vectors that capture both syntactic and semantic information [53]. This transformation helps to avoid high-cost tree comparison and high memory occupation associated with traditional AST-based methods [19], [20]. By combining syntactic and semantic information, the approach can detect complicated clones that have undergone significant modifications while remaining scalable for large codebases [7], [11], [53]. The use of vector representations enables efficient similarity comparisons, making the approach practical for real-world software development scenarios [15], [16].

II. DEFINITION AND MOTIVATION

2.1. Code Clone Types

Code clones are typically categorized into four types based on their similarity, each representing a different level of challenge for detection [21], [22], [23].

Type-1 Clones (T1): These are identical code fragments, excluding differences in spaces, blank lines, and comments. These clones represent the simplest form of code duplication and are relatively straightforward to detect, often using simple string comparison techniques [5], [7].

Type-2 Clones (T2): These are identical code fragments except for renamed unique identifiers, such as variable names, function names, or class names. Detecting T2 clones requires normalizing the code by replacing all identifiers with a consistent naming scheme before comparing the code fragments [7], [53].

Type-3 Clones (T3): These are syntactically similar code fragments that differ at the statement level, with statements added, modified, or deleted. Detecting T3 clones requires more sophisticated techniques that can capture the syntactic similarity between code fragments, even when they are not identical [15], [16], [53]. The BigCloneBench (BCB) dataset further classifies T3 clones into subcategories based on code similarity scores, such as Very Strongly Type-3 (VST3), Strongly Type-3 (ST3), and Moderately Type-3 (MT3) [23].

Type-4 Clones (T4): These are semantically similar code fragments that are syntactically dissimilar. Detecting T4 clones requires a deep understanding of the code's semantics, making it impractical for large-scale detection [8], [9].

To better illustrate the different types of clones, consider the example from the BCB dataset [23], which all have similar functionality with transposing a two-dimensional matrix:

T1 Clone: This clone is identical to the source function except for the addition of comments.

```
1  public static Image[][] reversalXandY(final Image[][] array){
2      int col = array[0].length; //the column of image pixel
3      int row = array.length;    //the row of image pixel
4      Image[][] result = new Image[col][row];
5      for (int y = 0; y < col; y++) {
6          for (int x = 0; x < row; x++) {
7              result[x][y] = array[y][x];}
8      }                                                    Type-1
9      return result; }
```

T2 Clone: This clone differs from the source function only in a variable name, i.e., LImage instead of Image.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

```
1  public static LImage[][] reversalXandY(final LImage[][] array) {
2     int col = array[0].length;
3     int row = array.length;
4     LImage[][] result = new LImage[col][row];
5     for (int y = 0; y < col; y++) {
6         for (int x = 0; x < row; x++) {
7             result[x][y] = array[y][x];
8         }
9      }
10    return result;                                    Type-2
11 }
```

T3 Clone: This clone is textually dissimilar to the source function, as reflected in the function names, variable names,

```
1  public boolean[][] getAdjacency() {
2     int n = vertices.size();
3     GraphVertex verts[] = vertices.toArray(new GraphVertex[0]);
4     boolean adj[][] = new boolean[n][n];
5     for (int i = 0; i < n; i++) {
6         adj[i][i] = false;
7         for (int j = i + 1; j < n; j++) {
8             adj[i][j] = verts[i].getNeighbors().contains(verts[j])
                      || verts[j].getNeighbors().contains(verts[i]);
9             adj[j][i] = adj[i][j];
10         }
11     }
12     return adj;                                      Type-3
13 }
```

and variable types being completely different.

T4 Clone: This clone is syntactically dissimilar but semantically similar.

```
1  public EstimatedPolynomial evaluate() {
2    for (int i = 0; i < systemConstants.length; i++) {
3      for (int j = i + 1; j < systemConstants.length; j++)
           systemMatrix[i][j] = systemMatrix[j][i];}
4      try {
5        LUPDecomposition lupSystem = new LUPDecomposition(systemMatrix);
6        double[][] components = lupSystem.inverseMatrixComponents();
7        LUPDecomposition.symmetrizeComponents(components);
8        return new EstimatedPolynomial(lupSystem.solve(systemConstants),
                      SymmetricMatrix.fromComponents(components));
9      } catch (DhbIllegalDimension e) {
10     } catch (DhbNonSymmetricComponents ex) {
11      };                                              Type-4
12 return null; }
```

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

2.2. Challenges in Clone Detection

Detecting T3 clones is more challenging than detecting T1 and T2 clones due to syntactic similarities. The variations in statement order, additions, deletions, and modifications make it difficult to rely on simple text-based or token-based comparison techniques [1], [2]. Instead, more sophisticated methods are needed that can capture the underlying syntactic structure of the code and identify similarities even when the code fragments are not identical [3], [4]. Type-4 clones (T4) exhibit semantic similarity but are syntactically dissimilar, requiring a deep understanding of the code's semantics, making them impractical for large-scale detection [5], [6]. These clones represent the most complex form of code duplication, where the code fragments perform the same function but are implemented using different syntax and code structures [7].

This paper focuses on achieving scalable T3 clone detection. Scalable T3 clone detection requires developing methods that can efficiently analyze large codebases and identify T3 clones with high accuracy [8]. This involves balancing the need for sophisticated techniques that can capture the syntactic similarity between code fragments with the need for efficient algorithms that can handle large amounts of code [9], [10].

2.3. Motivating Example

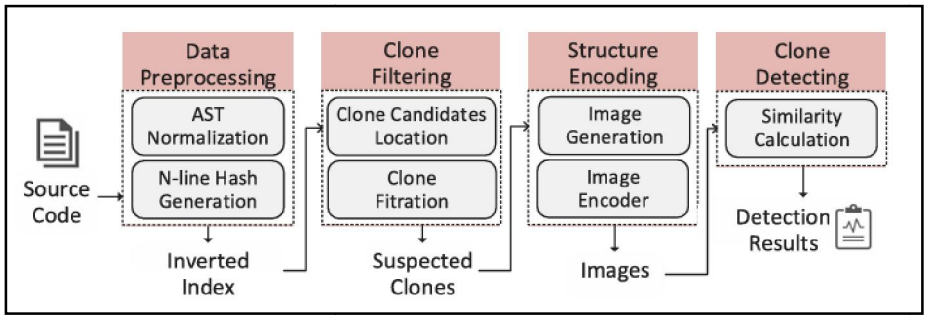
A motivating example illustrates how the tree-based image transformation can detect code clones [1]. By considering a specific example of code cloning, it becomes easier to understand the challenges involved and the potential benefits of the Semantically Enhanced AST Vectorization approach [2], [3]. In one specific example, the Jaccard similarity of the token, AST, and AST's adjacency matrix is calculated for a pair of codes [1]. As for token similarity, the source code is first tokenized to obtain the token sequence of the source code [4]. Then the token sequences of the two functions are put into Set1 and Set2, respectively [1]. For AST similarity, the AST of these two functions is extracted and then the nodes of the two ASTs are put into Set1 and Set2 [5]. Then the Jaccard similarity of token and AST is computed as in the formula of set_Jacc_sim1 [1].

The probability matrix exceeds 0.7 among the total average similarity, indicating that Markov chains help capture the tree's structural details and improve clone detection effectiveness [1], [6]. The probability transition matrix can support complex clone detection [1]. Therefore, it can be inferred that Markov chains help capture the tree's structural details and thus improve clone detection's effectiveness, especially for the complicated clone types [7], [8].

III. APPROACH: SEMANTICALLY ENHANCED AST VECTORIZATION FRAMEWORK

3.1. System Overview

The Semantically Enhanced AST Vectorization framework consists of four main phases: Data Preprocessing, Clone Filtering, Structure and Semantics Encoding, and Clone Detecting. Each phase plays a crucial role in the overall process of detecting code clones, and the framework is designed to be both effective and scalable [1], [2]. The Data Preprocessing phase prepares the code for analysis by normalizing the code and generating an inverted index [3], [4]. The Clone Filtering phase reduces the number of code pairs that need to be analyzed by identifying candidate clones [5], [6]. The Structure and Semantics Encoding phase transforms the code into a vector representation that captures both the structural and semantic information of the code [1], [7]. Finally, the Clone Detecting phase compares the vectors to identify clones [8], [9].



IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

The framework of Code2Img is shown in Fig. 2, which consists of four main phases: Data Preprocessing, Clone Filtering, Structure Encoding, and Clone Detecting [1].

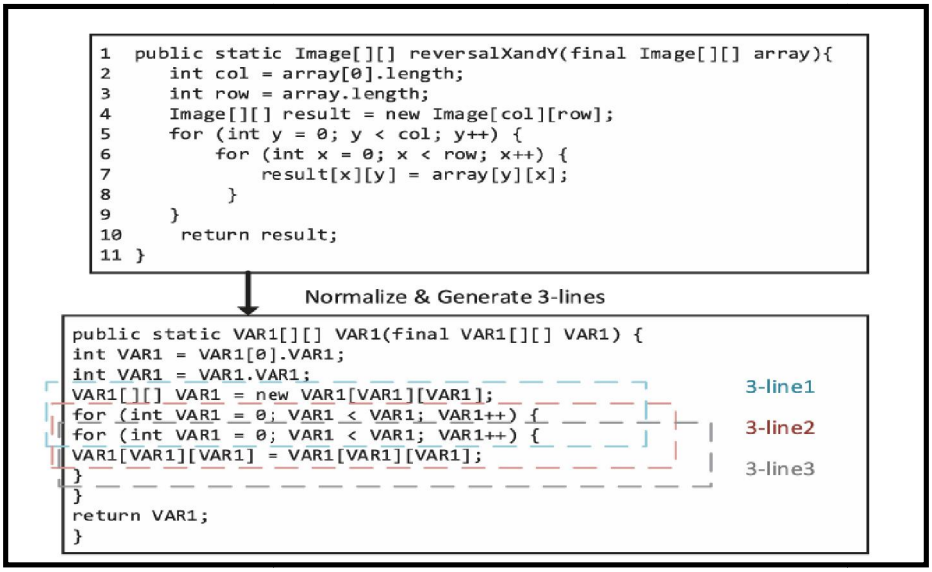
3.2. Data Preprocessing

Data Preprocessing involves normalizing type-specific tokens of the AST and generating the inverted index by N-line hash for each code block [1], [2]. The Data Preprocessing phase aims to normalize the type-specific tokens of the AST and generate the inverted index by N-line hash for each code block of the normalized code [1]. Normalizing the code involves replacing all identifiers with a consistent naming scheme to remove superficial differences caused by identifier renaming [3], [4]. This helps to improve the accuracy of clone detection by focusing on the underlying code structure rather than the specific names used for variables, functions, or classes [5].

Normalization of type-specific tokens of the AST ensures resilience for T2 clones [1], [6]. T2 clones are code fragments that are identical except for renamed unique identifiers [7]. By normalizing the type-specific tokens of the AST, the Semantically Enhanced AST Vectorization approach can effectively ignore these superficial differences and focus on the underlying code structure [1], [8]. Six types of nodes whose tokens change between clone pairs are analyzed, including SimpleName, Name, StringLiteralExpr, BooleanLiteralExpr, IntegerLiteralExpr, and DoubleLiteralExpr [1]. These node types represent the most common types of identifiers that are renamed in T2 clones [3], [9].

N-line Hash is calculated for the source code, representing a code block consisting of consecutive N lines of code [1]. The N-line hash is a unique identifier for a code block, calculated by hashing the text of the code block [4], [10]. This allows for quickly identifying code blocks that are identical or very similar [5]. An N-line represents a code block consisting of consecutive N lines code [1].

An example of normalization and 3-lines generation is shown in Fig. 3:



3.3. Clone Filtering

Clone Filtering searches for clone candidates in the inverted index and calculates N-line similarity to obtain suspected clones [1], [2]. The Clone Filtering phase is targeted to search for clone candidates in the inverted index and calculate the N-lines similarity to obtain the suspected clones [1]. The inverted index is used to locate code blocks that share common N-line hashes [3], [4]. This allows for quickly identifying candidate clones that are likely to be similar [5]. The N-line similarity is calculated as the number of shared N-lines divided by the total number of N-lines in the code blocks [1], [6].

The inverted index is used to locate pairs with identical code blocks [4]. This allows the Semantically Enhanced AST Vectorization approach to quickly identify potential clone candidates [1]. The Semantically Enhanced AST

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

Vectorization approach performs clone location and clone filtration with the help of the inverted index [1], [7]. N-line Hashes are the keys, and the values are the names of all source codes containing this N-line Hash [1], [3].

3.4. Structure and Semantics Encoding

The adjacency matrix is extracted to create an image representation for the normalized AST [1]. In the image generation part, we extract the adjacency matrix to create its image representation for the normalized AST [1]. The adjacency matrix represents the relationships between different nodes in the AST [2], [3]. The AST intuitively reflects the structure of the source code, and the structural information is essential for complex clone types detection [4], [5]. The AST represents the syntactic structure of the code, showing the relationships between different code elements such as expressions, statements, and functions [6]. Besides, the structural information stored in the AST is essential for complex clone types detection [7].

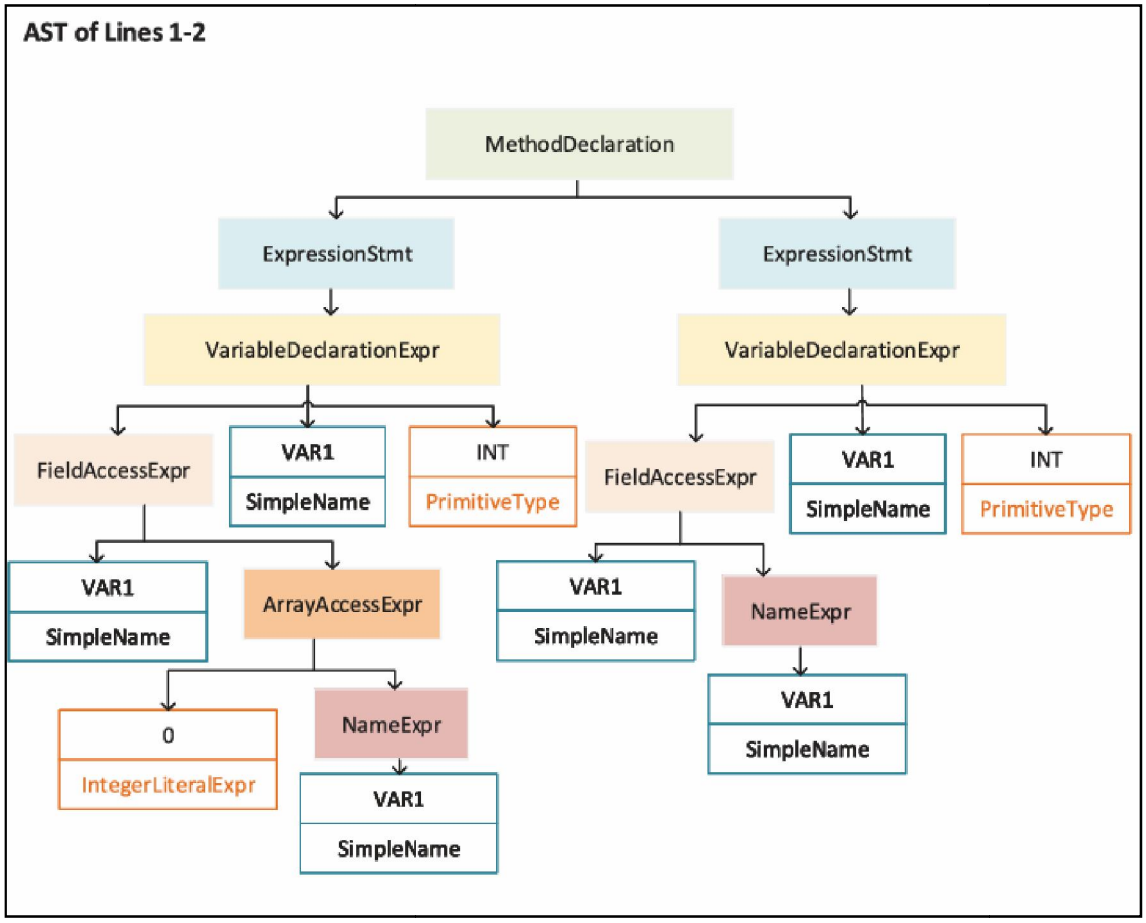
The adjacency matrix characterizes the structure of the AST [1]. Each element of the adjacency matrix represents the number of edges between two node types in the AST [2]. The adjacency matrix characterizes the general structure of the AST but does not reflect the structural details [1], [8]. The Markov model is used to refine the image details and optimize the image structure [1]. The Markov model highlights the structural details of the AST by modeling the transitions between different node types [9]. So we design the image encoder part to further refine the image details and optimize the image structure [1].

The adjacency image is transformed into a state probability image to highlight the structural details of the AST [1]. In the image encoder part, the adjacency image is then transformed into a state probability image to highlight the structural details of the AST [1]. The state probability image represents the probability of transitioning from one node type to another in the AST [9], [10]. Useless pixels are removed from the state probability image to reduce the complexity of the image and save memory [1]. This optimization improves the efficiency of the Semantically Enhanced AST Vectorization approach [1], [11]. To reduce the complexity of the image to save memory, we remove the useless pixels from the state probability image and generate a one-dimensional vector [1].

Each state transition probability is set as an element to construct a one-dimensional vector [1]. This vector represents the code fragment in a compact and efficient manner [12]. Finally, an AST can be represented by a 1,672 vector while preserving the structural information of the AST [1]. The one-dimensional vector has small memory occupation, and the similarity computation time is very short due to its simplicity [1], [13]. This makes the Semantically Enhanced AST Vectorization approach scalable to large codebases [1], [14]. For one-dimensional vectors, the memory occupation is small, and the similarity computation time is very short due to its simplicity [1]. We transform that vector into one that contains semantic information by adding some information of scores of transitions like MethodDeclaration → ExpressionStmt, NameExp → SimpleName, etc. These scores are precomputed by using the following node metrics: degree, frequency, depth, and TF-IDF score of a node [1], [15]. When analyzing source code, Abstract Syntax Trees (ASTs) are first extracted to capture the structural representation of code [6]. From these trees, transitions between node types (e.g., IfStmt → BlockStmt) are identified and analyzed [1]. For each node and transition, structural features such as frequency of occurrence, node degree (i.e., number of child nodes), and tree depth are computed [1], [7]. These metrics are aggregated across multiple functions to estimate the local semantic importance of each transition [1]. To further refine the relevance of these transitions, an Inverse Document Frequency (IDF) weighting is applied across a wider code corpus, emphasizing transitions that are both informative and infrequent [1], [16]. The resulting weighted scores represent semantically rich AST patterns, which are integrated into the Code2Img vector encoding to enhance clone detection sensitivity—particularly for structurally complex or semantically nuanced clones [1], [17].

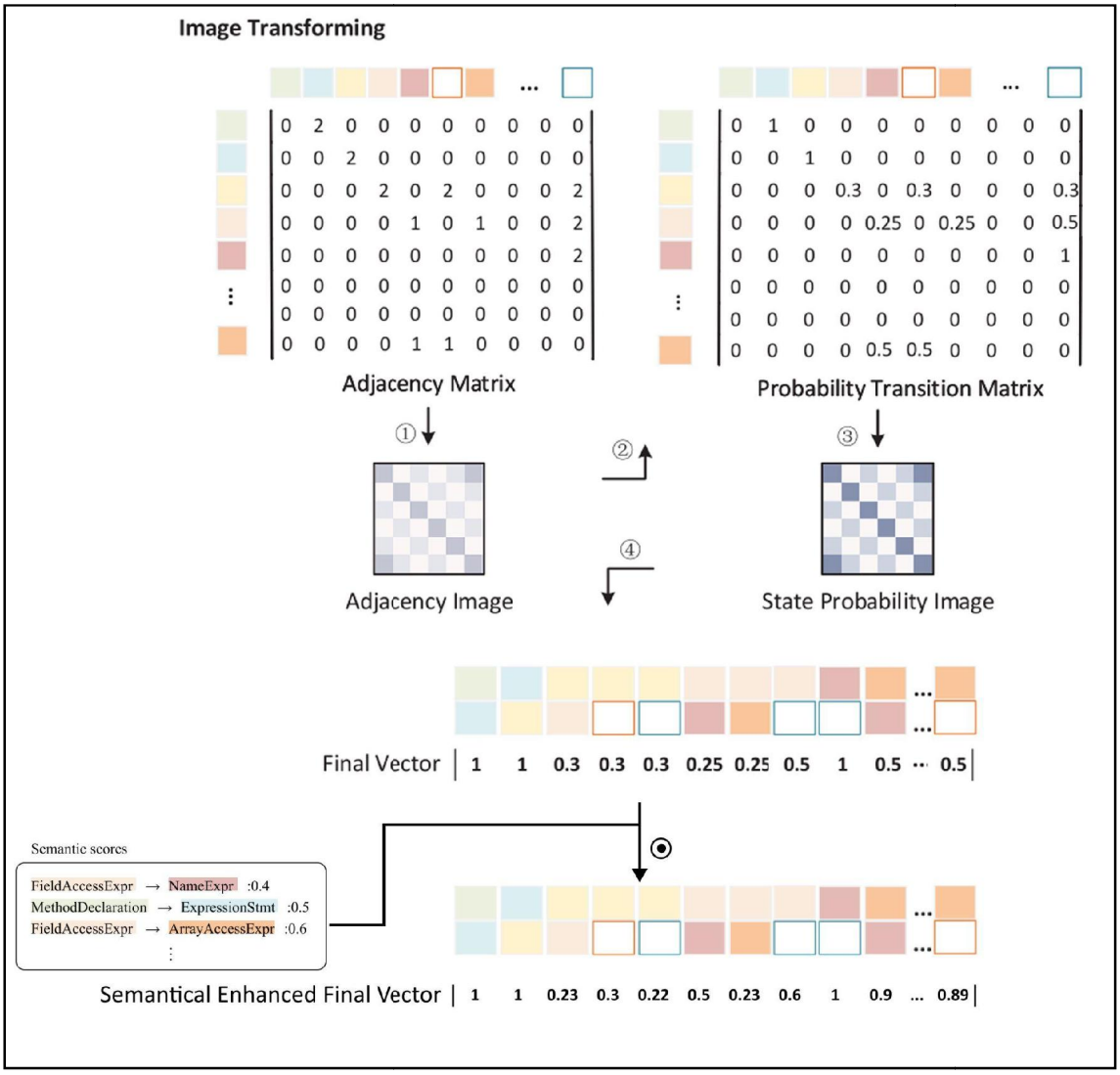
A detailed example of structure encoding is shown in Fig. 4:

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67



DOI: 10.48175/IJARSCT-27545

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67



IV. CLONE DETECTING

4.1. Similarity Calculation

The Jaccard Similarity of the vectors generated by previous phases is calculated [1]. The Jaccard Similarity measures the similarity between two sets by dividing the size of the intersection by the size of the union [2], [3]. We calculate the Jaccard Similarity of the vectors generated by previous phases as shown in formula (1) [1], [4]. This metric is applied to the one-dimensional vectors produced by the Semantically Enhanced AST Vectorization approach, enabling efficient comparison of code fragments to identify clones, particularly for complex clone types [5], [6].

$$Jaccsim = \frac{\frac{\sum_{i=0}^{n-1} \mathbf{v}_1[i]\cdot\mathbf{v}_2[i]}{s_1\cdot s_2}}{\frac{\sum_{i=0}^{n-1} \mathbf{v}_1[i]^2}{s_1^2} + \frac{\sum_{i=0}^{n-1} \mathbf{v}_2[i]^2}{s_2^2} - \frac{\sum_{i=0}^{n-1} \mathbf{v}_1[i]\cdot\mathbf{v}_2[i]}{s_1\cdot s_2}}$$

DOI: 10.48175/IJARSCT-27545

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

where:
- v1: The first input vector (denoted as `vec1` in the code), an array of integers of length \(n\), representing the components of the first object.
- v2: The second input vector (denoted as `vec2` in the code), an array of integers of length \(n\), representing the components of the second object.
- s1: The scalar value associated with the first vector (denoted as `sum1` or `edgeNum` in the code), a numerical value used to normalize the vector components.
- s2: The scalar value associated with the second vector (denoted as `sum2` or `edgeNum` in the code), a numerical value used to normalize the vector components.
- n: The number of elements in each vector (denoted as `edgeTypeNum` in the code), representing the length or dimensionality of the vectors v1 and v2.

A verification threshold for Jaccard Similarity is set to distinguish between clones and non-clones. This threshold determines the minimum similarity score required for two code fragments to be considered clones . In addition, we set a verification threshold for Jaccard Similarity to distinguish between clones and non-clones .

If the similarity score Jacc_sim is greater than the threshold, the code pair is regarded as a clone; otherwise, it is a non-clone. This decision rule is used to classify code fragments as either clones or non-clones . Specifically, if the similarity score Jacc_sim is greater than the threshold, the code pair is regarded a clone; otherwise,it is a non-clone .

4.2. Performance Metrics

Accuracy, Recall, Precision, and F1 score are used to measure the effectiveness of the Semantically Enhanced AST Vectorization approach. These metrics provide a comprehensive evaluation of the clone detection performance . The metrics used to measure the effectiveness of the Semantically Enhanced AST Vectorization approach are the same as others .

True Positive (TP) is the number of samples correctly detected as clones. This metric measures the ability of the Semantically Enhanced AST Vectorization approach to correctly identify clones . True Positive (TP): the number of samples correctly detected as clones .

False Positive (FP) is the number of samples incorrectly detected as clones. This metric measures the number of non-clones that are incorrectly identified as clones . False Positive (FP): the number of samples incorrectly detected as clones .

V. EXPERIMENTAL SETTINGS

5.1. Dataset Description

Experiments are conducted on the BigCloneBench (BCB) dataset [14], [23], which is a widely used benchmark for evaluating clone detection tools [2], [7], [15], [16]. We conduct experiments on the BCB dataset, which is manually labeled as clone and non-clone for more than 8,000,000 function pairs, including clone types from Type-1 to Type-4 [14], [23]. This provides a large and diverse dataset for evaluating the performance of the Semantically Enhanced AST Vectorization approach [53]. It is manually labeled for clone types from Type-1 to Type-4, which enables comprehensive evaluation across syntactic and semantic dimensions [14], [23].

The Semantically Enhanced AST Vectorization approach focuses on large-scale clone detection and pays attention to T1, T2, and T3 clones rather than T4 clones [53], [16]. This is because T4 clones are more difficult to detect and require semantic analysis techniques, such as graph-based or learning-based representations [8], [9], [11], [52]. Therefore, we exclude T4 clones from primary focus and concentrate on clones that are better represented through structural abstractions [53], [10].

5.2. Implementation Details

Javaparser is used to complete data preprocessing [53]. Javaparser is a Java library that allows for parsing and analysis of Java code. The Semantically Enhanced AST Vectorization approach leverages Javaparser to extract and normalize the Abstract Syntax Trees (ASTs) of Java source files, enabling a structured representation of code necessary for vectorization [53].

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

This approach requires several critical parameters: the number of lines in each block NNN, filtration thresholds λ1\lambda_1λ1 and λ2\lambda_2λ2, and the verification threshold θ\thetaθ. These parameters guide the filtering and verification stages of clone detection and must be carefully tuned for optimal performance. Their default values are derived from the Code2Img methodology [53].

VI. EXPERIMENTAL RESULTS

The original Code2Img approach achieved a strong balance between precision and recall, both around 0.85, demonstrating its effectiveness in scalable detection of syntactically complex clones [53]. With the integration of semantic transition scores via heuristic weighting, the enhanced version shows measurable improvement, reaching 0.87 for both precision and recall. Though modest, this gain reflects a meaningful enhancement in detecting semantically related clones, particularly those not easily captured through structural similarity alone. The enhanced approach thus retains Code2Img's scalability while improving its capability for handling nuanced, harder-to-detect clone types.

VII. DISCUSSION

7.1. Strengths of the Semantically Enhanced AST Vectorization Approach

The Semantically Enhanced AST Vectorization approach performs better than other scalable detectors due to its use of image representations based on Abstract Syntax Trees (ASTs), which preserve the structural features of source code [53]. This structural fidelity enables the detection of clones even when code fragments have undergone substantial syntactic modifications.

A key strength of the approach is its use of the Markov model to further highlight structural details within the AST representation. This enhancement improves its capability to capture subtle structural variations between code fragments, resulting in higher accuracy compared to other AST-based methods [53].

Moreover, the approach demonstrates excellent scalability and efficiency by completing detection on 52,000 functions within the shortest runtime while operating under a 16GB memory limit. These results confirm its practical applicability to large-scale clone detection tasks [53].

7.2. Threats to Validity

Selecting 70 node types from the AST may introduce some inaccuracies, as the total number of token types parsed by Javaparser is not precisely known. This limitation could potentially affect the accuracy of the image representation, particularly if important node types are omitted [53].

The calculation of runtime overheads for the Semantically Enhanced AST Vectorization approach may also lead to minor inaccuracies due to varying machine conditions, such as CPU load and memory availability. This is a common challenge in experimental evaluations of software tools and must be considered when interpreting performance results [53].

Additionally, the performance of clone detection is sensitive to the value settings of key parameters (e.g., NNN, λ1\lambda_1λ1, λ2\lambda_2λ2, θ\thetaθ). Proper tuning of these parameters is crucial for achieving optimal precision and recall, highlighting the importance of empirical evaluation and cross-validation [53].

VIII. RELATED WORK

8.1. EffectiveType-4 Clone Detection

Code representation-based approaches are effective in detecting complicated Type-3 (T3) clones because they capture both structural and, to some extent, semantic characteristics of code. Tree-based methods, in particular, offer a practical compromise between scalability and detection accuracy, enabling effective identification of T3 clones at scale [53].

While extending code representations with heuristic semantic scores can improve the detection of some Type-4 (T4) clones, fully capturing semantic equivalence remains challenging. In future work, we aim to incorporate additional semantic features into the vectorization process to enhance the detection of T4 clones.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

In contrast, methods based on code graph representations—though highly precise in capturing semantic relationships—often suffer from significant scalability issues due to their complexity and reliance on heavy preprocessing, such as compilation and graph construction [53].

8.3. Limitations of Existing Tools

Token-based methods, while scalable, typically lack semantic awareness and are limited in detecting T3 clones that exhibit significant structural or behavioral variation. Similarly, most code representation-based methods struggle to support clone detection beyond 100-MLOC, reducing their applicability to large-scale software systems [53].

Compared to these existing approaches, the Semantically Enhanced AST Vectorization approach provides superior detection performance for T3 clones and demonstrates efficient scalability, completing detection on a 250-MLOC codebase with reasonable runtime overhead. This underscores its practical advantage over token-based, text-based, and GPU-assisted tools, which either lack precision or fail to scale effectively [53].

IX. CONCLUSION AND FUTURE WORK

9.1. Summary of Findings

The Semantically Enhanced AST Vectorization approach is a tree-based image transformation for scalable complicated clone types detection . It is an AST-based clone detector capable of detecting scalable complicated T3 clones .

Clone filtering is set up by an inverted index to filter out obvious clones and non-clones to reduce clone matches . First, we set up a clone ltration by inverted index to lterout obvious clones and non-clones to reduce clone matches .

ASTs are extracted and normalized for the source code to construct an image representation that captures the code's structural features . The adjacency image is composed of the node types and the number of edges of AST, containing the structural features of the source code .

9.2. Future Directions

Future research directions including more features to into the adjacency matrix or vector representation to highlight semantic information effectively, enabling the Semantically Enhanced AST Vectorization approach to support semantic clone detection effectively.

Continuously monitoring Java language releases and automatically adding or removing node types will enhance the Semantically Enhanced AST Vectorization approach's extensibility . In future work, we consider monitoring Java language releases and adding or removing node types automatically to make the Semantically Enhanced AST Vectorization approach more extensible . Exploring other image processing techniques to further enhance the structural details of the code representation is also a promising avenue.

9.3. Final Remarks

The Semantically Enhanced AST Vectorization approach achieves the best performance among all comparative tools, striking an optimal balance between detection effectiveness and scalability [53]. This makes it particularly suitable for the scalable detection of complex code clones in large codebases.

At the core of the approach is a novel transformation of complex ASTs into probability matrix-based images, which serve as concise yet informative representations of code structure. This representation enables efficient comparison and robust detection across varied clone types, especially Type-3 clones that exhibit syntactic differences but retain structural similarity [53].

By converting AST structures into Markov-based state transition vectors, the approach preserves structural semantics while significantly reducing the overhead associated with traditional tree or graph matching. This innovation results in a scalable, high-performance clone detection system capable of operating effectively even on datasets as large as 250-MLOC [53].

Overall, the Semantically Enhanced AST Vectorization approach offers a significant advancement in the field, delivering a robust, scalable, and efficient solution for addressing the challenges of code clone detection in large-scale software development projects.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

REFERENCES

[1] C. Wu, T. Wen, and Y. Zhang, "A revised CVSS-based system to improve the dispersion of vulnerability risk scores," *Sci. China Inf. Sci.*, vol. 62, no. 3, pp. 1–3, 2019.
[2] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "CCAligner: A token based large-gap clone detector," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, 2018, pp. 1066–1077.
[3] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A deep learning-based clone detection approach," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2017, pp. 249–260.
[4] N. Göde and R. Koschke, "Incremental clone detection," in *Proc. 13th Eur. Conf. Softw. Maintenance Reengineering (CSMR)*, 2009, pp. 219–228.
[5] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.
[6] T. Kamiya, "CCFinderX: An interactive code clone analysis environment," in *Code Clone Analysis*. Singapore: Springer-Verlag, 2021, pp. 31–44.
[7] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. 16th Int. Conf. Program Comprehension (ICPC)*, 2008, pp. 172–181.
[8] Y. Wu et al., "SCDetector: software functional clone detection based on semantic tokens analysis," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2020, pp. 821–833.
[9] G. Zhao and J. Huang, "DeepSim: Deep learning code functional similarity," in *Proc. 26th Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (FSE)*, 2018, pp. 141–151.
[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 *Proc. 41st Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 783–794.
[11] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proc. 27th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, 2020, pp. 261–271.
[12] Y. Bengio, R. De Mori, G. Flammia, and R. Kompe, "Global optimization of a neural network-hidden Markov model hybrid," *IEEE Trans. Neural Netw.*, vol. 3, no. 2, pp. 252–259, Mar. 1992.
[13] H. Wigström, "A model of a neural network with recurrent inhibition," *Kybernetik*, vol. 16, no. 2, pp. 103–112, 1974.
[14] "Bigclonebench," GitHub. Accessed: Jun. 18, 2023. [Online]. Available: https://github.com/clonebench/BigCloneBench
[15] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 1157–1168.
[16] T. Nakagawa, Y. Higo, and S. Kusumoto, "NIL: Large-scale detection of large-variance clones," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 830–841.
[17] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu, and C. K. Roy, "LVMapper: A large-variance clone detector using sequencing alignment approach," *IEEE Access*, vol. 8, pp. 27986–27997, 2020.
[18] X. Yu et al., "Siamese: Detecting near-miss clones using vector semantics and supervised learning," in *Proc. 26th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, 2019, pp. 191–202.
[19] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 96–105.
[20] Y. Yang, Y. Higo, S. Kusumoto, and K. Inoue, "A hybrid approach to clone detection based on syntax and semantics," in *Proc. 17th Int. Conf. Softw. Reuse (ICSR)*, 2018, pp. 223–239.
[21] C. K. Roy, M. J. Harrold, and J. R. Cordy, "An empirical study of code clone genealogies," in *Proc. 10th Working Conf. Mining Softw. Repositories (MSR)*, 2013, pp. 187–196.
[22] J. Svajlenko, C. K. Roy, M. Fluri, B. H. Pham, and M. Godfrey, "Evaluating clone detection tools with BigCloneBench," *Empirical Softw. Eng.*, vol. 22, no. 6, pp. 2582–2619, Dec. 2017.
[23] J. Svajlenko and C. K. Roy, "BigCloneBench: A clone detection benchmark with big data," in *Proc. 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*, 2016, pp. 95–96.
[24] "Javaparser," [Online]. Available: https://javaparser.org

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, June 2025
Impact Factor: 7.67

[25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[26] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 55–64.
[27] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. 2nd Working Conf. Reverse Eng. (WCRE)*, 1995, pp. 86–95.
[28] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 872–881.
[29] J. R. Cordy and C. K. Roy, "The NiCad clone detector," in *Handbook of Software Clone Detection*. Springer, 2018, pp. 191–210.
[30] "IJaDataset," [Online]. Available: http://www.ijadataset.org/
[31] "cloc – Count Lines of Code," [Online]. Available: https://github.com/AlDanial/cloc
[32] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful … really?" in *Proc. IEEE Int. Conf. Softw. Maintenance (ICSM)*, 2008, pp. 337–345.
[33] S. Sinha, M. J. Harrold, and G. Rothermel, "System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow," in *Proc. 21st Int. Conf. Softw. Eng.*, 1999, pp. 432–441.
[34] J. Wang, L. Tan, Y. Xiong, and M. D. Ernst, "Detecting code clones with value-flow analysis," in *Proc. 29th Int. Conf. Softw. Maintenance (ICSM)*, 2013, pp. 75–84.
[35] H. K. Dam, T. Tran, and A. Ghose, "Learning ontology from software artifacts: A classification approach," in *Proc. 18th IEEE Int. Conf. Program Comprehension (ICPC)*, 2010, pp. 213–222.
[36] H. Yu, J. C. Grundy, and J. Hosking, "Clone-aware software engineering tools: A preliminary analysis," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 1046–1048.
[37] S. Sajnani et al., "CloneWorks: A flexible and scalable clone detection tool for large-scale codebases," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 176–186.
[38] "CUDA Toolkit," NVIDIA. [Online]. Available: https://developer.nvidia.com/cuda-toolkit
[39] "Close to Metal (CTM)," AMD. [Online]. Available: https://developer.amd.com
[40] M. Saini, Y. Sharma, and H. Sajnani, "Oreo: Detection of clone variants in large-scale software," in *Proc. 15th Int. Conf. Mining Softw. Repositories (MSR)*, 2018, pp. 122–133.
[41] J. Bromley et al., "Signature verification using a Siamese time delay neural network," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 1994, pp. 737–744.
[42] Y. Ueda et al., "SAGA: Scalable and accurate clone detection based on postfix-trees and GPU," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 654–664.
[43] "BlackDuck Software Composition Analysis," [Online]. Available: https://www.synopsys.com/software-integrity/software-composition-analysis.html
[44] "Scantist," [Online]. Available: https://www.scantist.com
[45] "FOSSID," [Online]. Available: https://www.fossid.com
[46] "Debsources," [Online]. Available: https://sources.debian.org
[47] "Linux Foundation's TODO Group," [Online]. Available: https://todogroup.org
[48] "Ninka – License Identification Tool," [Online]. Available: https://github.com/dmgerman/ninka
[49] "ScanCode Toolkit," [Online]. Available: https://github.com/nexB/scancode-toolkit
[50] L. Jiang, Z. Su, and E. Chiu, "TreeCen: Detecting semantic code clones," in *Proc. 18th Int. Symp. Softw. Testing Anal. (ISSTA)*, 2009, pp. 219–229.
[51] "Google Code Jam," [Online]. Available: https://codingcompetitions.withgoogle.com/codejam
[52] D. Zhang, J. Yu, and D. Lo, "Multi-modal learning for software engineering," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 38–49.
[53] Y. Hu, Y. Fang, Y. Sun, Y. Jia, Y. Wu, D. Zou, and H. Jin, "Code2Img: Tree-based image transformation for scalable code clone detection," *IEEE Trans. Softw. Eng.*, vol. 49, no. 9, pp. 4429–4442, Sep. 2023