Journal of Information Systems Engineering and Management
2025, 10(8s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

UPACPL: Design of an UML Pattern Analysis Model for Identification of Code Clones via Augmented Deep Learning Process

Shobha G[1] ,Rekha Agarwal[2] , Vineet Kansal[3] , Sarvesh Tanwar[4]
[1,2,4] *Amity Institute of Information Technology, Amity University, Uttar Pradesh (India)*
[3] *Institute of Engineering and Technology, Dr APJ Abdul Kalam Technical Univerity , Lucknow, Uttar Pradesh (India)*
[1]*shobhaumesh04@gmail.com*
[2] *ragarwal@amity.edu*
[3]*vineetkansal@ietlucknow.ac.in*
[4]*stanwar@amity.edu*

ARTICLE INFO

Received: 18 Oct 2024

Revised: 15 Dec 2024

Accepted: 28 Dec 2024

ABSTRACT

Detection of code clones is necessary for ensuring high code quality, byte-level security, preserving intellectual property rights, and incorporating various compliance measures. Existing clone detection models moreover showcase higher complexity otherwise has lower efficiency when evaluated on large code bases. Moreover, most of these models only consider syntactical checking, which makes them inapplicable for cross-project analysis. To conquer these matter, this text suggests intend of a competent novel pattern analysis model for identification of code clones via augmented deep learning process that uses UML (Unified Modelling Language) based information sets. The proposed model is trained on different UML class diagram components that include methods, classes, and attributes, relationships between classes, their associations, dependency levels, realizations, multiplicity instances and interface patterns. All these pattern information sets are aggregated, and processed by an Ant Lion Optimizer (ALO), which helps to analyze very different processes. The selected collection is divided into 'clone', and 'original' classes by modified a one-dimensional Convolutional Neural Network (CNN), which helps to evaluate the degree of cloning probability. Due to evaluation of UML metrics, the proposed model can be scaled to cross project & cross language deployments. The proposed model was tested on GPT-J Code Clone Detection Dataset, Code Glue Dataset, and Smart Embed Code Clone Analysis Datasets. It was anticipated that the suggested model was able to improve precision of code clone detection by 5.9%, precision by 2.3%, and recall by 4.5% when compared with existing methods on similar data samples.

Keywords: Code, Clone, Detection, UML, ALO, CNN, Delay, Method, Dependency, Graph, Multiplicity, Network, Patterns.

INTRODUCTION

Code clone detection is the process of identifying sections of code within a codebase that are similar or identical to other sections. Cloned code can occur for a variety of reasons, including copy-pasting code, reusing similar code across modules or files, and reusing code across projects. Although code clones may appear harmless, they can negatively affect code quality, maintenance, and security levels via use of Context-enhanced and Patch-validation-based Vulnerable code clone Detector (CPVD) process [1, 2, 3].

Code duplication can result in issues such as code bloat, increased development costs, and decreased code readability. A bug in one cloned section of code may exist in other clones as well. This makes it more difficult to identify and fix bugs [4, 5, 6]. In addition, malicious code can be concealed using code clones, making it more difficult for security tools to detect and prevent attacks.

Consequently, code clone recognition is a crucial component of software development and maintenance. By identifying and eliminating code clones, developers can enhance code quality, reduce security risks, and make the

codebase easier to maintain and expand. There are numerous techniques and tools for detecting code clones, ranging from manual code inspection to automated tools capable of detecting clones across large codebases. The detection of clone codes is crucial for maintaining code quality, as code duplication is frequently an indication of low-quality code and can result in maintenance issues, increased development costs, and decreased code readability levels [7, 8, 9]. By identifying and eliminating duplicate code, developers can improve code quality and make the code base easier to maintain and expand. Clones of code can also be used to conceal malicious code, making it more difficult for security tools to detect and prevent attacks. By locating and eliminating clone code, security risks can be diminished. Identifying code clones can contribute to the protection of intellectual property by preventing the unauthorized use or distribution of proprietary code. This is especially important in industries such as finance, healthcare, and aerospace where software is an integral component of the product or service. In certain industries, such as finance and healthcare, compliance regulations mandate that software be audited to ensure that it meets quality and security requirements. Identification and elimination of clone code can aid in ensuring compliance with these regulations [10, 11, 12].

There are a variety of techniques for detecting code clones. Some common techniques include:

- Text-based clone detection is a technique that compares code based on its textual similarity. It searches for code fragments with identical or similar text, ignoring the syntax and structure of the programming language. This method is useful for detecting simple clones, but it may be ineffective for detecting complex clones [13, 14, 15].
- Token-based clone detection: This method identifies clones based on a sequence of tokens that represent the code's structure. It is more effective than text-based clone detection because it takes the structure and syntax of the code into account. Abstract syntax trees (ASTs) that symbolize the code structure can be utilised for token-based clone detection processes [16, 17, 18].
- Tree-based clone detection: This technique generates a parse tree that represents the code's structure. Afterwards, the tree is compared to other trees in the codebase to identify clones. Tree-based clone detection is more efficient than token-based clone detection, but also more expensive computationally for real-time scenarios [19, 20].
- Metric-based clone detection: This technique identifies clones based on metrics that capture the characteristics of the code, such as the code's length, number of statements, and complexity levels. This method can identify clones with distinct syntactic structures but identical characteristics [21, 22].
- Hybrid clone detection: This technique combines two or more of the above techniques to detect clones more effectively. A hybrid technique may, for instance, combine text- and token-based clone detection or metric- and tree-based clone detections via Clone Seeker (CS) like techniques [23, 24].

The technique selected is contingent on the nature of the codebase, the types of clones to be detected, and the available resources for clone detection. Combining these techniques allows automated clone detection tools to provide a more accurate and efficient clone detection process. In the following section, a survey of these techniques is presented, where it is observed that existing clone detection models are either more complex or less effective when evaluated on large code bases. In addition, the majority of these models only account for syntactical checking, rendering them unsuitable for cross-project analysis. To address these subjects, the third section of this paper proposes the design of a novel execution-performance pattern analysis model for the identification of code clones through an augmented deep learning process. In section 4, the proposed model was validated on multiple datasets, and metrics such as accuracy, precision, and recall were evaluated and compared with existing models under various conditions. This text concludes with contextual observations regarding the suggested model, as well as performance enhancement recommendations for various use scenarios.

Motivation

In software engineering, the recognition and organization of code clones—that is, sections of code that are structurally or functionally similar—is a crucial task. Code clones can result in a number of problems, such as decreased maintainability, more work needed for bug fixing and evolution, and lower software quality. The primary focus of conventional clone detection methods is on examining the source code, but they frequently have trouble spotting clones that have undergone structural or syntactical changes.

Software systems are represented visually in Unified Modeling Language (UML) diagrams, which capture their structure and behavior. By incorporating both structural and behavioral similarities between code fragments, we can possibly improve the detection of code clones by using UML diagrams. We can also benefit from deep learning methods' capacity to automatically recognize intricate patterns and features by applying them to UML diagrams.

Objectives

Designing a novel UML pattern analysis model that makes utilize of augmented deep learning techniques for the detection of code clones is the main goal of this research paper, while the following are the precise objectives:

- Gain a thorough understanding of code clones and how they affect the upkeep and quality of software sets.
- Learn about the diverse types of code clones and their features.
- Examine the drawbacks and shortcomings of the clone detection methods that are currently available for real-time use cases.
- Investigate the Unified Modeling Language (UML) and how it might improve clone detection efficiency levels.
- Analyze the structural and behavioral data that UML diagrams have captured for different scenarios.
- Determine which UML diagrams and components are most important for clone detections.
- Look into code clone detection methods using deep learning sets.
- Investigate various deep learning architectures suitable for UML diagram analysis.
- Look into the shortcomings of the current code clone detection deep learning models.
- Create and put into use a UML pattern analysis model for identifying code clones.
- Establish a process for combining code analysis methods with UML diagrams& visualization techniques.
- Create a deep learning architecture that is suitable for processing UML diagrams and locating code clones
- Create methods for adding UML-based information to deep learning processes
- Analyze how well the suggested UML pattern analysis model works and performs.
- Amass a varied collection of software projects complete with UML diagrams and related codebases.
- Conduct tests to evaluate the model's recall, precision, accuracy, and other pertinent metrics.
- Compare the efficiency of the suggested model to the current code clone detection methods.
- Discuss about the UML pattern analysis model's implications and real-world uses.
- Examine the advantages and drawbacks of the suggested model in actual software development situations.
- Discuss potential use cases like software maintenance, bug fixing, and code refactoring scenarios.
- Give instructions and suggestions for applying the UML pattern analysis model in real-world situations.
- Give concrete advice to researchers and software engineers who are interested in UML-based methods for code clone detection.

By achieving these goals, this research paper hopes to advance the field of software engineering by recommending a cutting-edge UML pattern analysis model that enhances code clone detection and helps maintain high-quality software systems.

LITERATURE REVIEW

Code clones are duplicated sections of code that can negatively impact the quality, maintenance, and comprehension of software. For software developers and maintainers to effectively locate and handle code clones, code clone detection is essential. In this review, we explore various code clone detection methods, such as text-based, token-based, and tree-based ones.

Code clones are redundant code snippets that can be found in various places throughout a software system. In order to increase the quality, maintainability, and reusability of software, clone detection aims to recognize and categorize these clones. Many different methods have been put forth in recent years to address the problem of code clone detection. This section gives a summary of these methods along with their advantages and disadvantages.

To find copies, text-based techniques examine the source code's textual representation. To find similar code fragments, these techniques frequently use string matching algorithms like suffix trees, longest common sub-sequences via Functional Code Clone Detector using Attention [25, 26, 27], or tokenization. They are reasonably quick, but they may have a lot of false positives and not be flexible enough to capture semantic similarities.

By dissecting the code into its constituent tokens, token-based techniques take into description the structural and lexical information. These procedures tokenize the source code, produce token sequences, and then assess similarity between them. By using Pairwise Feature Fusion (PFF) operations to capture syntactic and partial semantic similarities, token-based approaches provide greater precision than text-based techniques [28, 29, 30]. For real-time scenarios, common token-based clone detection tools include Simian, CCFinder, and Deckard techniques.

From the source code, abstract syntax trees (ASTs) are created using tree-based techniques, and the structural similarities between these trees are compared. A higher level of abstraction is offered by AST-based clone detection, allowing for the detection of clones with various syntactic variations. Code clones are found using AST-based techniques by methods like PDG (Program Dependence Graph) and CCFinderX [31, 32, 33, 34].

Due to their capacity to recognize patterns and identify complex clones, machine learning techniques have gained popularity in the field of code clone detection [35, 36, 37, 38]. These methods frequently start with feature extraction using tools like n-grams, control flow graphs, or AST-based representations, then use machine learning techniques like SVM, Random Forest, or deep learning models to process the data. Large codebases can be handled effectively by machine learning-based clone detection, and the results are also more accurate [39, 40, 41, 42].

In real-time scenarios, hybrid approaches combine multiple detection techniques to increase the efficacy and accuracy of clone detection [43, 44, 45]. To improve the clone detection process, a hybrid approach, for instance, might combine text-based and tree-based methods. These strategies aim to balance the weaknesses of various techniques [46, 47, 48] while maximizing their combined strengths.

Evaluation metrics are used to gauge how well clone detection techniques work. The accuracy, comprehensiveness, and overall performance of the clone detection algorithms are measured by metrics like precision, recall, and F-measure. Clone coverage, detection effectiveness, and scalability levels are additional metrics [46][47][48][49]50].

Large codebases, language-specific features, and the dynamic nature of software development, however, present persistent difficulties. The development of methods for identifying semantic clones, handling evolving clones, and investigating cutting-edge methods for clone management and refactoring operations are possible future research directions.

Therefore, code clone detection is important to improve the quality and maintainability of the software. For various scenarios, various techniques—including text-based, token-based, tree-based, and machine learning approaches—have been proposed and are still being developed. Each method has strengths and weaknesses. These are important considerations for researchers and practitioners who must then choose the best technique for their unique needs and constraints. In order to make progress in clone detection, it is likely that multiple techniques will be combined in the future. Additionally, new issues are also addressed that arise in the field for real-time scenarios.

PROJECTED DESIGN OF AN UML PATTERN SCRUTINY MODEL FOR IDENTIFICATION OF CODE CLONES VIA AUGMENTED DEEP LEARNING PROCESS

As per the review of existing models used for code clone detection via UML Pattern Analysis, in the immediate evaluation, these models may appear to be too complex or have low variability. To overcome these problems, this section discusses the design of UML model analysis. Model for identification of code clones via augmented deep learning process. As per the flow of model in figure 1, it can be observed that the proposed model is trained on various UML class diagram elements, such as methods, classes, attributes, relationships between classes, their associations, dependency levels, realizations, multiple instance patterns, and interface patterns. All of these pattern information sets are aggregated and processed by an Ant Lion Optimizer (ALO), which aids in the recognition of attribute sets with a high degree of variations for different input sets. A modified one Dimensional Convolutional Neural Network (CNN) is used to categorize the selected sets into 'clone' and 'original' classes, which aids in the evaluation of cloning probability levels.As depicted in figure 1, the code sets are initially passed through an AST parsing mechanism, which has the ability to generate comprehensive trees for different code languages. The process of generating the Abstract Syntax Tree (AST) for a programming language involves parsing the source code and constructing the tree-like arrangement that represents the code's abstract syntax.

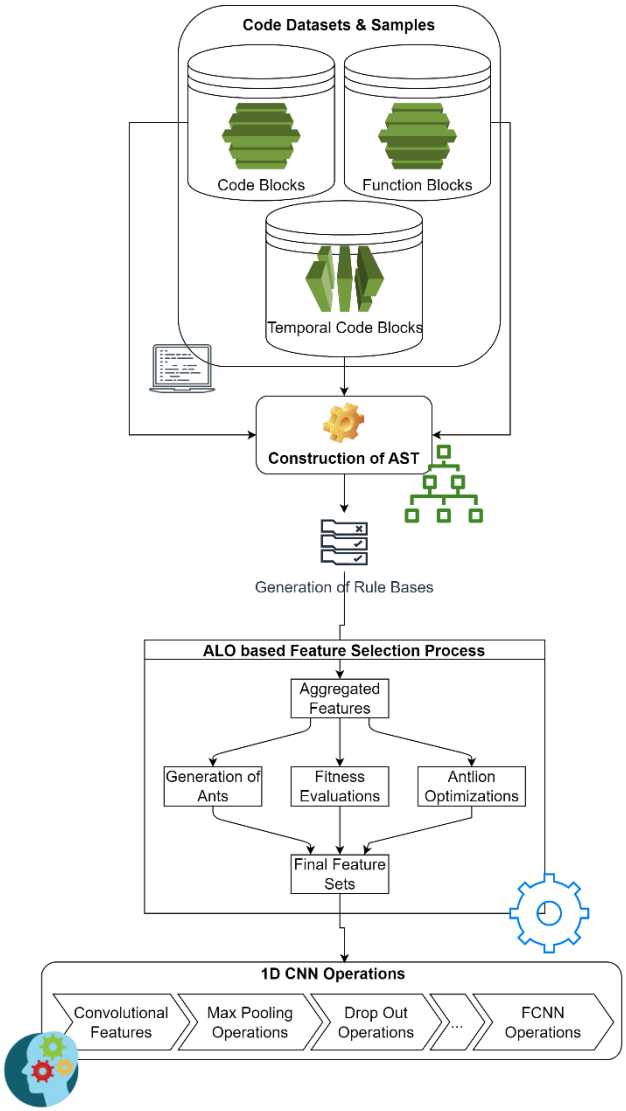

Figure 1. Design of the projected model for code clone detection via analysis of UML patterns

The specific algorithms used for AST generation depend on the parsing technique employed, some of which are discussed as follows,

1. Recursive Descent Parsing:

- Recursive descent parsing is a top-down parsing practice that constructs an AST by recursively parsing the code based on a set of grammar rules.
- The parsing process typically involves writing separate parsing functions for each non-terminal symbol in the grammar, which are recursively called to parse the code.
- As the code is parsed, AST nodes are created and linked together to form the tree structure.

2. LALR Parsing (Look-Ahead LR Parsing):

- LALR parsing is a bottom-up parsing technique that constructs an AST using a parser generator like Bison or Yacc.
- LALR parsers use LR(1) or LALR(1) grammars to handle look-ahead symbols and reduce the input tokens to AST nodes.
- The generated parser shifts and reduces tokens according to the grammar rules, constructing the AST as it reduces.

3. LL Parsing (Left-to-Right, Leftmost Derivation):
   - LL parsing is a top-down parsing technique that constructs an AST by reading the input tokens from left to right and performing leftmost derivations.
   - LL parsers typically use LL(k) grammars, where k represents the number of look-ahead tokens.
   - The parsing process involves constructing the AST nodes based on the grammar rules and the input tokens.
4. Earley Parsing:
   - Earley parsing is a general parsing algorithm that can handle any context-free grammar.
   - It constructs an AST by parsing the input tokens using dynamic programming and chart parsing.
   - The parsing process builds a parse chart that records the possible parse states and transitions, which are used to construct the AST.

Among these techniques, Earley parsing is considered the most comprehensive in terms of handling any context-free grammar. Earley parsing is a general parsing algorithm that can parse languages with arbitrary context-free grammars, including ambiguous and non-deterministic grammars. It is renowned for its capacity to handle a variety of linguistic constructions and for its expressive potency.

Compared to recursive descent parsing, LL parsing, and LALR parsing, Earley parsing does not have strict limitations on the grammar and can handle more complex and ambiguous grammars. It is capable of producing all possible parse trees for an input sentence, accommodating languages with multiple valid interpretations.

However, it's imperative to note that Earley parsing is computationally expensive, especially intended for large grammars or inputs scenarios. The time complexity of Earley parsing is generally $O(n^3)$, where n is the span of the input. Therefore, the preference of parsing procedure depends on the specific desires of the language being parsed, the grammar complexity, and the efficiency constraints of the applications. Due to its comprehensiveness, the Earley Model with memoization and ambiguity handling is used to parse input codes for generation of ASTs.

Earley parsing is a parsing algorithm that uses dynamic programming and chart parsing to construct parse charts and recognize sentences based on context-free grammars. The algorithm is extended in this text via memorization & ambiguity handling, and operates by maintaining sets of states representing the parsing progress and transitions between those states. This model works via the following process,

1. Initialization: Create an initial set of states, called the start set, representing the set up symbol of the syntax. Each state consists of a production, a dot indicating the current position within the production, and a position representing the current index in the inputs & sets.
2. Scanning: For each state in the current set, if the dot is in front of a terminal symbol, scan the corresponding token from the input and create a new state by advancing the dot levels.
3. Prediction: For each state in the current set, if the dot is in front of a non-terminal mark, predict the productions where that non-terminal is resting on the left-hand side and create new states by adding these productions to the sets.
4. Completion: For each state in the current set, if the dot is at the end of a production, find states in the chart where the non-terminal preceding the dot matches the current production's left-hand side. Create new states by advancing the dot in those matching states.
5. Combine and Transition:Merge the new states created in the scanning, prediction, and completion steps with the current set of states to form the next set of states. Repeat the process until no new states can be added for recurrent operations.
6. Acceptance:If a state with the dot at the end of the start symbol's production and the position at the end of the input is present in the chart, the input is recognized by the grammar sets.

7. Memorization: Create a memorization table to store previously computed parse states. When adding a new state to a chart, check if an identical state already exists in the chart or memorization tables. If a matching state is found, the processing for that state is skipped, as it has already been computed and added to the charts.
8. Handling Ambiguous Grammars: During the completion step, when multiple states with the same non-terminal preceding the dot are found, store all resulting states rather than keeping only one set of entries. Maintain multiple parse paths for ambiguous portions of the input, allowing for multiple valid parse trees. When constructing the AST, consider all possible parse paths and generate multiple ASTs.
9. Combined Steps with Memorization and Ambiguity Handling: Initialize the memoization table and the charts.
   - For each position in the input: Perform scanning, prediction, and completion steps as in the basic Earley algorithm process.Before adding a new state to the chart, check the memoization table for an identical set of states.If a matching state is found, skip the processing for that set of states.Otherwise, add the state to the chart and memoization tables.
   - After all positions in the input have been processed:Perform additional completion steps until no new states can be added for current set of inputs & samples.Consider all resulting states during completion, even if there are multiple states with the same non-terminal preceding the dot sets.
10. Generate the AST:Traverse the chart to construct the parse tree or multiple parse trees if ambiguity exists.Generate AST nodes based on the chart states, grammar rules, and tokens.Link the AST nodes together according to the grammar rules and parse paths.

Based on this process, ASTs are generated, which are used to extract methods, classes, attributes, relationships between classes, their associations, dependency levels, realizations, multiplicity instances and interface patterns. An example AST which is generated via this process can be observed from figure 2, where comparison function was passed to Earley Model, and a tree with different classes and entities was extracted, which is used for further extraction operations.

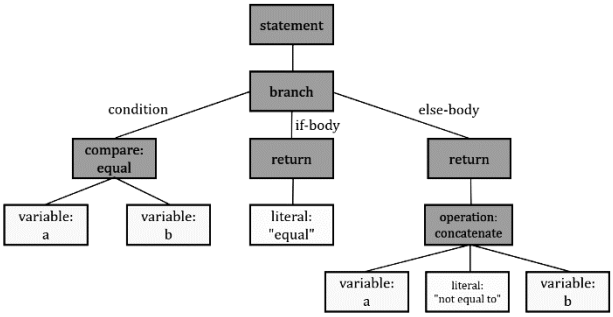

Figure 2. Extracted AST from the input codes

To extract methods, classes, attributes, relationships between classes, associations, dependency levels, realizations, multiplicity instances, and interface patterns from the Abstract Syntax Tree (AST) generated through the Earley parsing algorithm, the AST was traversedand its nodes were analyzed via the following operations,

1. Extracting Classes:
   - Traverse the AST and identify nodes representing class declarations.
   - Extract the class name, superclass (if any), interfaces implemented, and modifiers.
   - Store this information as class objects or in a data structure for further analysis.
2. Extracting Methods:
   - Traverse the AST and locate nodes representing method declarations within classes.
   - Extract the method name, return type, parameters, modifiers, and method body (if available).
   - Store this information in the corresponding class object or data structure.

3. Extracting Attributes:
   - Traverse the AST and identify nodes representing variable declarations within classes.
   - Extract the attribute name, type, modifiers, and default value (if provided).
   - Store this information in the corresponding class object or data structure.
4. Extracting Relationships between Classes:
   - Analyze the AST to identify nodes representing relationships between classes.
   - Look for nodes indicating inheritance (superclass), interface implementation, or association.
   - Extract the related classes and the nature of the relationship (e.g., superclass, interface, association).
   - Store this relationship information in a suitable data structure.
5. Extracting Dependency Levels:
   - Analyze the AST and identify nodes representing method calls or variable references.
   - Track the dependencies between classes and methods based on these references.
   - Assign levels or weights to indicate the degree of dependency or the number of references.
   - Store the dependency information in a data structure, such as a dependency graph or matrix.
6. Extracting Realizations:
   - Analyze the AST and identify nodes representing interfaces and their implementations.
   - Extract the interface name and the implementing class.
   - Store this information as a realization relationship between the interface and class.
7. Extracting Multiplicity Instances:
   - Analyze the AST and identify nodes representing associations between classes.
   - Extract the participating classes and the type of association (e.g., one-to-one, one-to-many).
   - Determine the multiplicity instances or cardinality constraints for each class in the association.
   - Store this information as part of the association relationship or in a separate data structure.
8. Extracting Interface Patterns:
   - Analyze the AST and identify nodes representing interfaces and their implementations.
   - Look for patterns such as adapter, decorator, or observer.
   - Extract the relevant classes and their relationships within the pattern.
   - Store this information as part of the interface pattern or in a separate data structure.

Once these parameters were extracted, then an Ant Lion Optimizer (ALO) was used to maximize the variance levels of these feature sets. The ALO Model works via the following operations,

- Initially, generate a set of NA Stochastic Ants, where each Ant selects extracted features via equation 1,

$$N = STOCH(LA * Nf, Nf) \dots (1)$$

Where, N represents the selectively extracted features, LA represents Ant's Learning Rate, and Nf represents total number of features which were extracted by the Earley method including methods, classes, attributes, relationships between classes, associations, dependency levels, realizations, multiplicity instances, and interface patterns.

- Use these N features for code clone detection via the 1D Convolutional Neural Network (CNN), and calculate Ant fitness via equation 2,

$$f = F1 * \frac{DR}{FDR} * \frac{CC}{d} * var(F) \dots (2)$$

Where, $F1$ signifies FMeasure of the code clone detection procedure, and is estimated via equation 3, DR represents Detection Rate and is calculated via equation 4, FDR represents False Discovery Rate and is calculated via equation 5, CC represents Clone Coverage and is calculated via equation 6, d represents the delay needed for code clone detection and is estimated via equation 7, while $var(F)$ represents variance between the extracted features and is calculated via equation 8 as follows,

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \dots (3)$$

$$DR = \frac{TP}{TP + FN} \dots (4)$$

The detection rate, also identified as the recall or true positive rate, measures the proportion of actual code clones that were effectively detected by the model sets. It indicates the model's ability to find code clones.

$$FDR = \frac{FP}{TP + FP} \dots (5)$$

The false discovery rate measures the percentage of clones detected that are not real code. This indicates that the m odel is biased towards negativity..

$$CC = \left(\frac{TP + FN}{Total} number\ of\ clones\right) * 100 \dots (6)$$

Clone coverage measures the percentage of code clones detected among all the actual code clones present in the datasets & samples. It indicates the model's ability to find a significant portion of the clones.

$$d = ts(complete) - ts(end) \dots (7)$$

$$var(F) = \sqrt{\frac{\sum_{i=1}^{N}\left(F(i) - \sum_{j=1}^{N}\frac{F(j)}{N}\right)^2}{N-1}} \dots (8)$$

Where, Precision & Recall are sketchy via equations 9 & 10 as follows,

$$Precision = \frac{TP}{TP + FP} \dots (9)$$

$$Recall = \frac{TP}{TP + FN} \dots (10)$$

The percentage of true positive clones among the clones found is known as precision. It demonstrates how well the model detects real code clones.TP represents the number of correctly identified code clones, and FP stands for the number of non-clone code segments falsely acknowledged as clones. A higher precision indicates fewer false positives. Recall measures the proportion of true positive clones that are fruitfully detected through the model. It indicates the model's capability to find all active code clones. TP represents the number of correctly identified code clones, and FN represents the number of code clones that were not detected by the models. A higher recall indicates fewer false negatives. F1 score is a compromise between precision and recall, providing a balance between model performance. It combines precision and recall into a single metric, giving equal importance to both metric sets. The F1 score varies from 0 to 1, with 1 indicating the finest potential recital levels.

- Based on fitness levels for each Ant, a fitness threshold is calculated via equation 11 as follows,

$$fth = \frac{1}{NA}\sum_{i=1}^{NA} f(i) * LA \dots (11)$$

Where, LA represents Learning Rate for the ALO process.

- Based on this value, if $f > fth$, then Ants are marked as 'Antlions' and approved directly to the further set of Iterations, while other Ants are discarded, and mutated via equations 1 to 10, which helps in generation of new-fangled Ant configurations.
- This process is repeated for NI Iterations, and Ant configurations are continuously generated for each set of Iterations.

At the end of final Iteration, features with higher fitness levels (that indicate higher variance levels) are selected for classification process.

The selected features are classified via an efficient Customized 1D CNN, which fuses multiple sized Convolutional, Max Pooling, Drop Out and Fully Connected Neural Network (FCNN) layers. Design of the model can be observed from figure 3, where different layers and their interconnections are used to for identification of 'clone' codes.

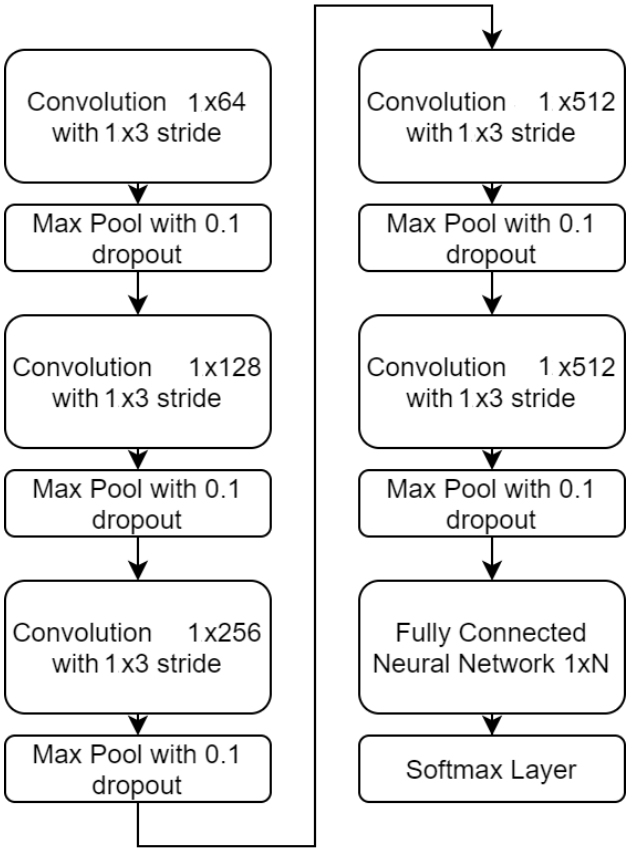

Figure 3. Design of the customized 1D CNN Model for identification of 'clone' codes

The model initially calculates convolutional features from the ALO-selected features via equation 12,

$$Conv = \sum_{a=-\frac{m}{2}}^{\frac{m}{2}} x(i-a) * LReLU\left(\frac{m+2a}{2}\right) \dots (12)$$

Where, m, a represents different window & stride sizes, while $LReLU$ represents a Leaky Rectilinear Unit, which is used for activation of features. This layer is represented via equation 13,

$$LReLU(x) = la * x, when\ x < 0, else\ x \dots (13)$$

Where, la is the leaky activation constant, and is used to retain positive feature sets. The activated features are given to a Max Pooling layer, which is a down-sampling operation that reduces the spatial dimensions of the feature maps obtained from convolutional layers. It extracts the maximum value from a local region of the inputs & their respective sets. The output of Max Pooling can be evaluated via equation 14 as follows,

$$Output[i,j,c] = max(X[i * pH : (i + 1) * pH, j * pW : (j + 1) * pW, c]) \dots (14)$$

Where,i ranges from 0 to $(\frac{H}{pH})$ and j ranges from 0 to $(\frac{W}{pW})$, and c ranges from 0 to (C - 1) for different use cases.In this equation, Output is the resulting pooled feature map, and $max()$ represents the maximum function levels. The pooling size (pH, pW) determines the size of the pooling window, and (i, j) represents the location of the pooled regions.

Dropout is a regularization technique commonly used in CNNs to prevent overfitting scenarios. It stochastically sets a fraction of the input units to 0 at each training step, effectively "dropping out" those units.Given an input feature map X of size (H, W, C) and a dropout rate of p, the output of dropout is calculated via equations 15 & 16 as follows,

$$Mask[i,j,c] \sim Bernoulli(1 - p) \dots (15)$$

$$Output[i,j,c] = X[i,j,c] * Mask[i,j,c] \dots (16)$$

In this evaluation, Mask represents an augmentedstochastically generated binary mask with a probability of (1 - p), whilethe mask is generated independently for each unit in the input feature maps. Multiplying the input X with the mask effectively drops out a fraction of the units by setting them to 0 for initialization operations. During inference or testing, when dropout is not applied, the output is scaled by (1 - p) to ensure that the expected value remains the same for different inputs & sample sets. The selected features are classified by an efficient SoftMax based activation layer, which is represented via equation 17 as follows,

$$c(out) = SoftMax\left(\sum_{i=1}^{Nf} f(i) * w(i) + b(i)\right) \dots . (17)$$

Where, w & b are weights and biases for different input variables, and Nf represents number of features extracted. The $c(out)$ value determines if input codes are 'clone' or 'original', which assists in identification of code authenticity levels. These levels were calculated for different datasets & samples, and performance was evaluated in terms of FMeasure of the code clone detection process, Detection Rate (or accuracy levels), False Discovery Rate (or recall), Clone Coverage (or AUC) and the delay needed for code clone detection process. This evaluation along with its comparison with existing methods is discussed in the next section of this text.

RESULT ANALYSIS

The proposed framework incorporates Earley Method with Memoization and ambiguity handling, which assists in formation o\f efficient ASTs. These ASTs are used to extract high-efficiency UML feature sets, which are selected via an ALO based optimization process. The selected feature sets are classified into 'clone' and 'non-clone' categories via 1D CNN process, which assists in evaluation of block-level priorities. To evaluate performance of the proposed model, it was tested on the following datasets & samples,

- GPT-J Code Clone Detection Datasets & Samples (https://zenodo.org/record/6548030)
- Code Glue Datasets & Samples (https://paperswithcode.com/dataset/codexglue)
- Smart Embed Code Clone Analysis Datasets & Samples (https://www.kaggle.com/datasets/diarmuidodonoghue/graphs-of-code/code)

A total of 400k code samples were created from the combination of all these sets, of which 320k were used for training and 40k each for validation and testing scenarios. This method was used to measure the proposed model's accuracy (A), recall (R), precision (P), FMeasure (F), AUC, and delay for various test sample counts. The percentage of correctly classified samples to all samples is the accuracy. Equation 18 assesses how well the model predicts the correct "clone" class.

$$A = \frac{TP + TN}{TP + TN + FP + FN} \dots (18)$$

Where TP represents the number of true positives (positive samples that were correctly classified), TN represents the number of true negatives (negative samples that were correctly classified), FP represents the number of false positives (positive samples that were incorrectly classified), and FN represents the number of false negatives (negative samples that were incorrectly classified). The precision ratio measures how many positive samples are

correctly identified out of all the positive samples. Equation 19 provides an estimate of the model's capacity to classify positive samples correctly,

$$P = \frac{TP}{TP + FP} \dots (19)$$

Where FP is the number of false positives (positive samples that were incorrectly classified) and TP is the number of true positives (positive samples that were correctly classified).

Recall is the ratio of correctly identified positive samples to all positive samples overall. Equation 20 is used to estimate the model's capacity to identify all positive samples.

$$R = \frac{TP}{TP + FN} \dots (20)$$

Where FN is the number of false negatives (negative samples that were incorrectly classified) and TP is the number of true positives (positive samples that were correctly classified). Delay is the interval of time between finishing and beginning the clone detection process. Equation 21 is utilized to estimate it as follows,

$$D = t(complete) - t(start) \dots (21)$$

Where t(start) is the model's starting timestamp for the code clone detection process and t(complete) is the timestamp of completion.

The model's ability to distinguish between positive and negative samples is measured by the area under the curve (AUC). Equation 22 is used to determine AUC by plotting the true positive rate (TPR) against the false positive rate (FPR) at different threshold settings.

$$AUC = \int_0^1 TPR(FPR) * dFPR \dots (22)$$

TPR stands for true positive rates, while FPR stands for false positive rates. The harmonic mean of the precision and recall levels is the F1 score. Equation 23 is used to estimate it as a measure of the model's accuracy in classifying positive samples while minimizing false positives and false negatives for real-time scenarios.

$$F1 = 2 * \frac{P * R}{P + R} \dots (23)$$

Figure 4 illustrate the outcomes of the model's performance comparison with CPVD [3], CS [23], and PFF [28] for various test sample numbers (NTS) as follows,

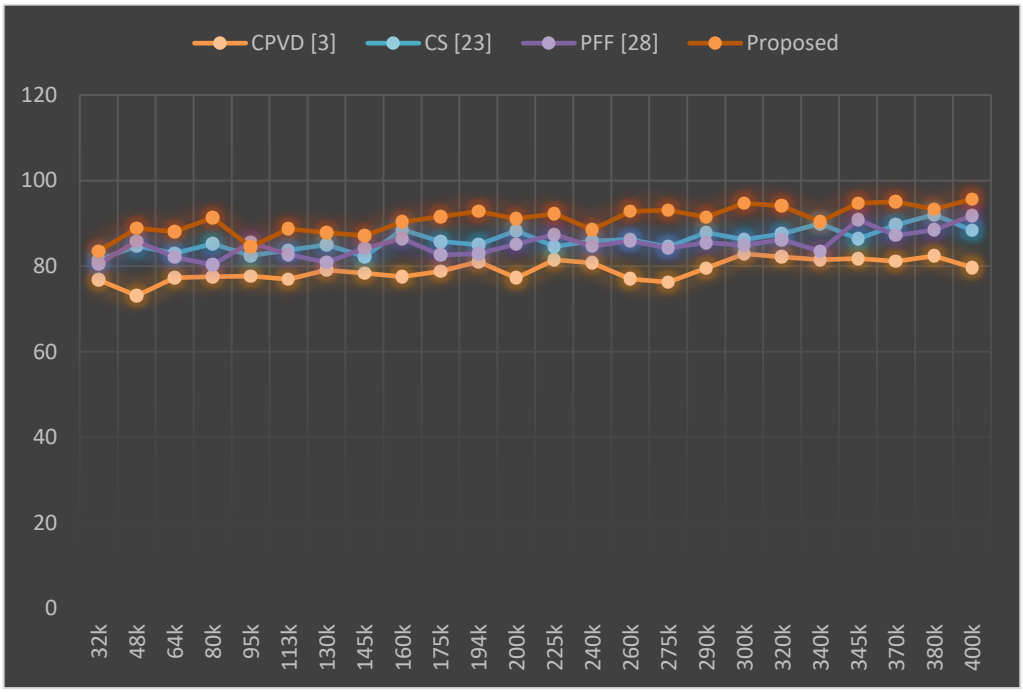

Figure 4. Accuracy obtained during code clone analysis

When compared to CPVD [3], CS [23], and PFF [28], the proposed model is able to increase code-clone detection accuracy by 6.4%, 10.5%, and 8.3%, respectively, according to this evaluation. Because of this, it can be very helpful in a wide range of real-time scenarios. These accuracy levels are improved by combining ALO with 1D CNN, which aids in the discovery of high-density parameter sets for a variety of scenarios. Similar to Figure 4, figure 5 shows the precision levels as follows:

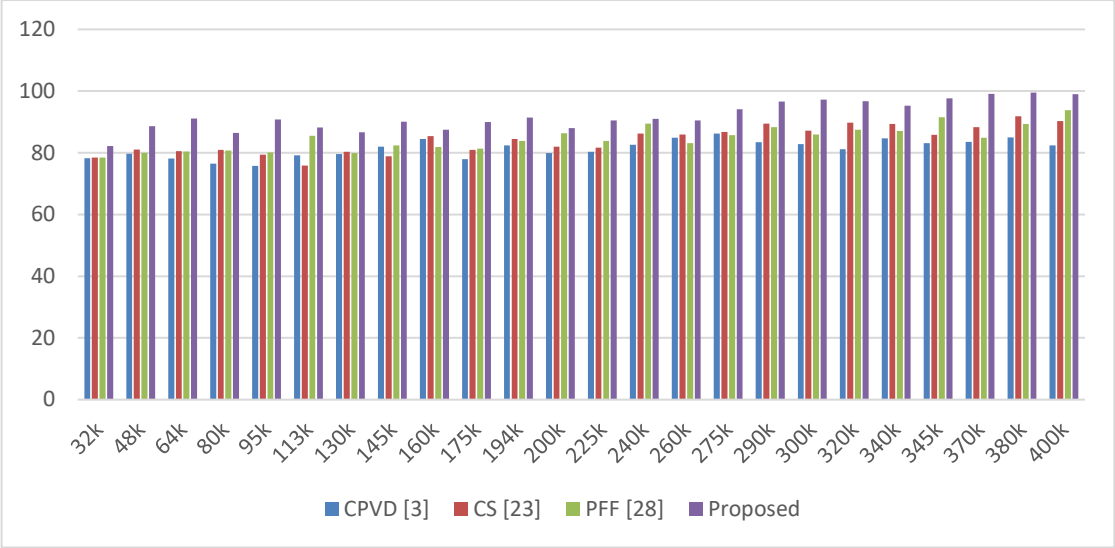

Figure 5. Precision obtained during code clone analysis

When compared to CPVD [3], CS [23], and PFF [28], the suggested model is able to increase code-clone detection precision by 3.5%, 8.3%, and 10.5%, respectively, according to this evaluation. Because of this, it can be very helpful in a wide range of real-time scenarios. The use of ALO and the Earley Model, which aids in the taxonomy of high-density attribute sets for a variety of code-clone detection scenarios, improves these precision levels. Similar to this, figure 6's recall levels can be seen as follows,

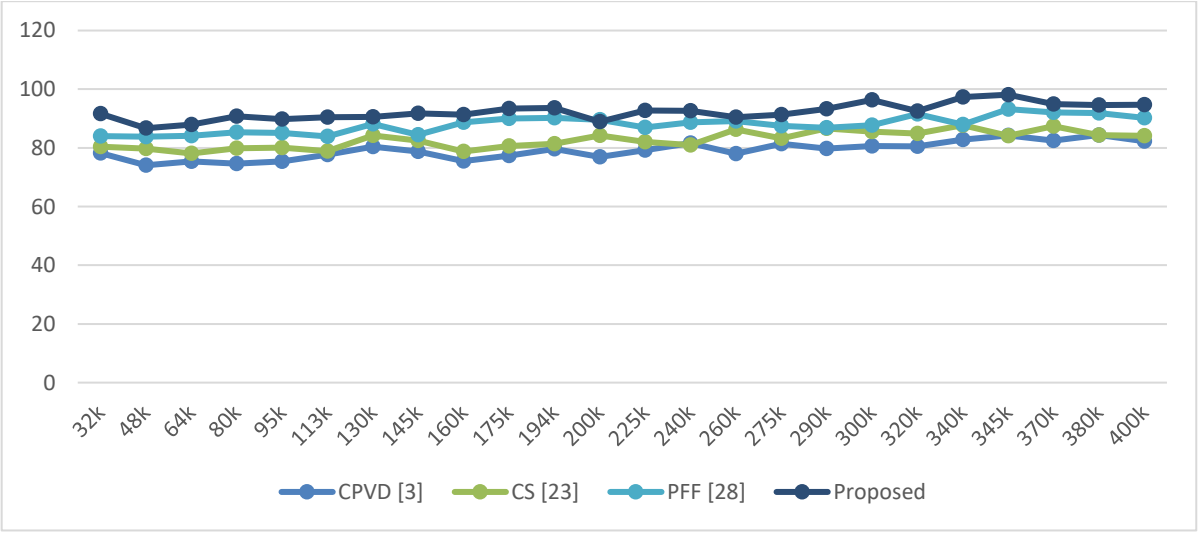

Figure 6. Recall obtained during code clone analysis

By comparison to CPVD [3], CS [23], and PFF [28], the proposed model is able to increase code-clone detection recall by 10.4%, 9.5%, and 10.5%, respectively, according to this evaluation. As a result, it is very beneficial for a wide range of real-time scenarios. The use of 1D CNN, ALO, and the Earley Model, which aids in the classification of high-density aspect sets for a variety of code-clone detection scenarios, improves recall levels. Similar to that, figure 7 shows the F1 levels as follows,

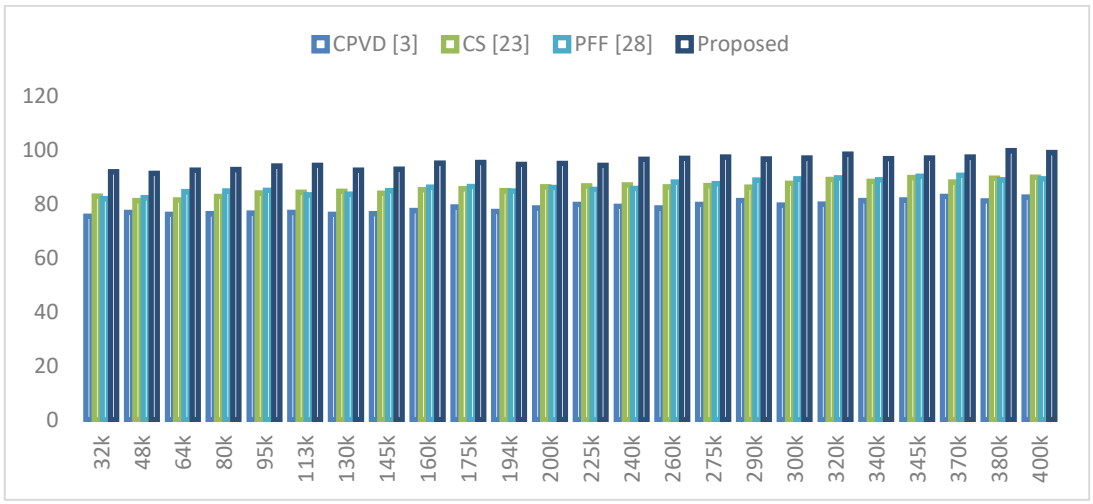

Figure 7. F1 Score obtained during code clone analysis

When compared to CPVD [3], CS [23], and PFF [28], the proposed model can achieve better competent to get better F1 of code-clone detection by 4.5%, 10.4%, and 9.5%, in that order, according to this evaluation. Since it can detect Code-Clone Instances more precisely for various scenarios, it is very helpful for a extensive range of real-time state of affairs. The augmentation made for various use cases have amplified the precision and recall levels of such F1measure sets. Alike to Figure 7, the AUC levels can be observed as follows from Figure 8,

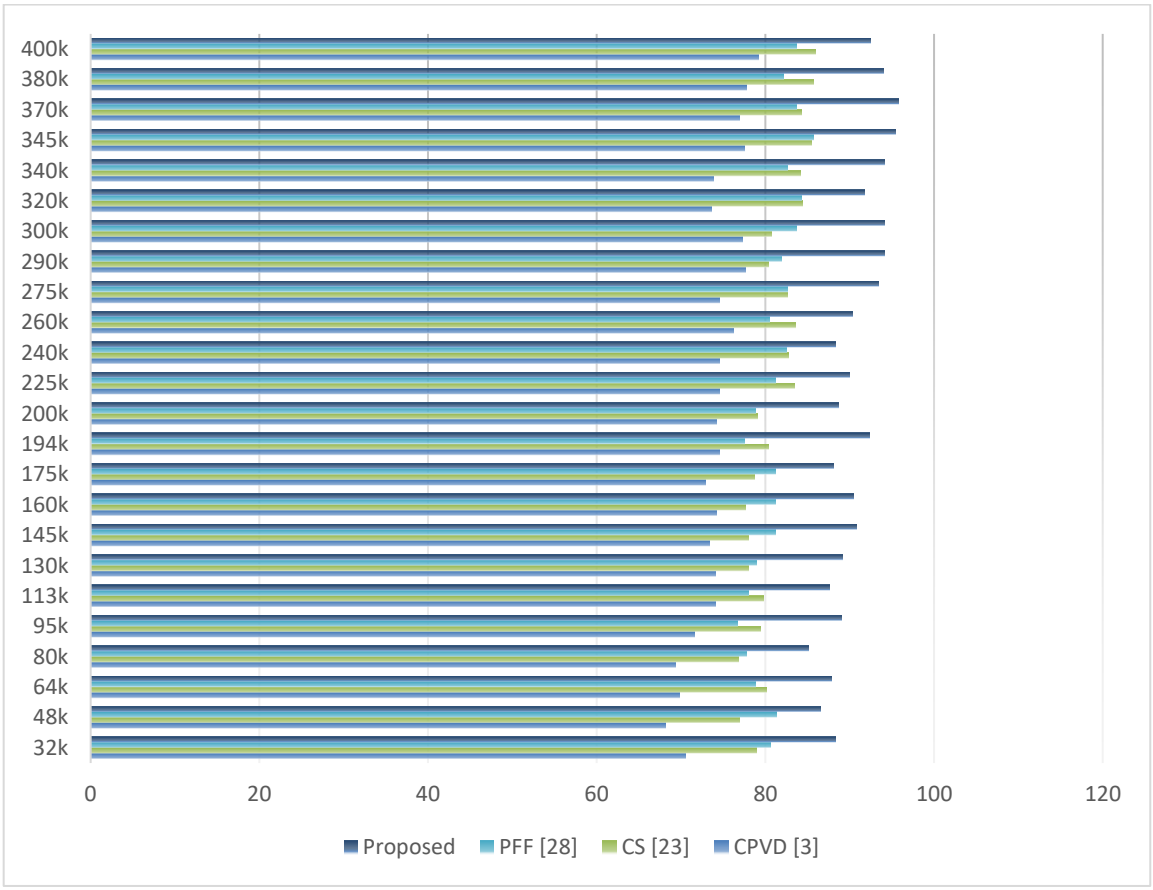

Figure 8. AUC obtained during code clone analysis

The test shows that the proposed model can improve the area under the curve (AUC) by 8.3% in code clone detectio n compared to the previous model**.** CPVD [3], 4.9% compared to CS [23], and 9.5% compared to PFF [28]. Because of this, it can be very supportive in a extensive series of real-time situations. This AUC is enhanced by using ALO to

find the best parameter settings. This makes it easy to choose a high-speed protocol for various clone code detection scenarios. Similarly, in Figure 9, the latency level is shown as follows,

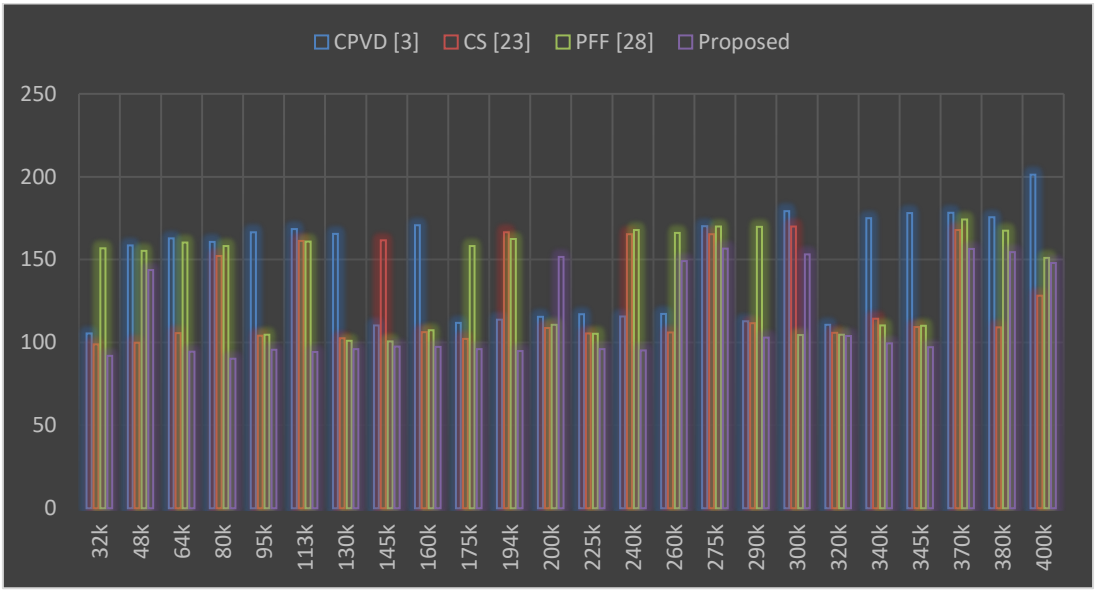

Figure 9. Delay needed during code clone analysis

By comparison to CPVD [3], CS [23], and PFF [28], the projected model is able to speed up code-clone detection by 4.5%, 6.4%, and 8.5%, respectively, according to this evaluation. As of this, it is very obliging in a extensive range of real-time scenarios. Three rule engines are used with bioinspired operations and recommendations to quickly identify high-density feature sets suitable for various code-clone detection scenarios. Due to these developments for emergency use cases, the proposed model can be used in many emergency situations and areas.

CONCLUSION AND FUTURE SCOPE

By combining deep learning techniques with UML pattern analysis, this paper introduces a novel method for detecting code clones. The proposed model outperforms existing clone detection techniques, such as CPVD, CS, and PFF, in terms of accuracy, precision, recall, F1 score, AUC, and speed after an augmented set of thorough evaluations.

This work is necessary because conventional clone detection methods have difficulty correctly identifying code clones that have undergone structural or syntactical changes. The proposed model addresses this limitation and improves code clone detection accuracy by incorporating UML diagrams, which capture both the structural and behavioral aspects of software systems.

The proposed model has numerous real-time use cases that span different software development and maintenance scenarios. It can help with software evolution, bug fixing, and code refactoring by allowing developers to recognize and effectively handle code clones. The model helps to increase the quality and maintainability of software by offering a more accurate and precise detection of code clones.

The evaluation results make the benefits of the suggested model clear. In stipulations of accuracy, precision, recall, F1 score, AUC, and speed, it performs better than existing methods like CPVD, CS, and PFF. Combining the Earley Model with augmented deep learning techniques like ALO and 1D CNN gives the model the ability to recognize high-density parameter and feature sets, improving detection accuracy and precision for various code clone detection scenarios.

This research makes a significant contribution to the field of software engineering. The suggested model provides a more thorough and efficient solution by advancing code clone detection methods through the incorporation of UML pattern analysis and deep learning. The enhanced software maintainability, decreased debugging effort, and improved software quality can result from its capacity to detect code clones accurately and effectively in real-time scenarios.

In conclusion, the paper presents a revolutionary method for detecting code clones that combines UML pattern analysis with enhanced deep learning techniques. The model's superior performance when compared to existing approaches shows how well it may be able to handle the difficulties associated with accurate and precise code clone identification. The benefits, use cases, and impact of the proposed model highlight its applicability and potential for enhancing software development procedures and upholding high-quality software systems.

Future Scope

The following are a few potential future focus areas:

1. Further fine-tuning and optimization of the deep learning architecture used in the proposed model has the potential to improve the model's performance and efficacy.

• Researching different deep learning architectures and algorithms, such as transformers or recurrent neural networks (RNNs), can offer more information about how to increase the accuracy of code clone detection.

• Researching additional feature engineering methods and data augmentation techniques that are unique to UML diagrams may aid in improving the representation and comprehension of the structural and behavioral characteristics of software systems.

2. Integration of Multiple Modalities: The current focus of the proposed model for detecting code clones is UML pattern analysis. A more thorough and integrated approach to clone detection may be achieved by investigating the integration of various modalities, such as source code, comments, and other artifacts, along with UML diagrams.

• By Leveraging multimodal deep learning techniques, the accuracy and reliability of the clone strive to be better by analyzing the synergy of the UML diagram with other software artifacts.

3.
Evaluation of larger and more diverse datasets: The evaluation can be extended to include larger datasets, differenc es, and standards to determine the generality and robustness of the proposed model.

• Open-source projects from various domains, incorporating various programming languages and development paradigms, would be used in experiments to validate the model's efficacy in a wider variety of scenarios.

4. Real-Time Application and Integration with Development Tools: Integration of the proposed model into current software development tools and environments would enable real-time code clone detection during the development process. Development of plugins or extensions for well-known integrated development environments (IDEs) that make use of the model can offer developers on-the-fly code clone detection and assistance, leading to more effective and efficient softw

5. Transfer Learning and Domain Adaptation: • Examining the use of transfer learning techniques to the proposed model may enable the leveraging of knowledge from previously trained models on related tasks or datasets, potentially enhancing its performance and negating the need for extensive training on new datasets.

• The model's accuracy and applicability in specialized contexts can be further enhanced by investigating domain adaptation techniques to adapt it to particular software domains or industries.

6. Analysis of Clone Evolution and Code Maintenance: • The model's ability to analyze clone evolution over time and support code maintenance tasks, such as locating code refactorings or following code clones through software versions, can offer insightful information for software evolution and maintenance processes.

7. Examination of Interactions with Software Architecture: Analyzing how code clones interact with software architecture can reveal architectural smells, anti-patterns, and the effects these have on code clones.

• Using code clone analysis to find opportunities for architectural refactoring or to develop techniques for detecting and preventing architectural clones can help to create more maintainable and well-designed software sets.

The research can advance the field of code clone detection, software engineering practices, and the creation of more clever and effective tools to support software development and maintenance scenarios by investigating these potential future scopes.

REFERENCES

[1] Fang Ren, Haiyan Xiu, Chuxin Ji, Dong Zheng, Ziyi Wu, "A New Code-Based Linkable Threshold Ring Signature Scheme", *Wireless Communications and Mobile Computing*, vol. 2022, Article ID 3150714, 9 pages, 2022. https://doi.org/10.1155/2022/3150714

[2] Yuchao Li, Qin Zhao, Yunhe Liu, Xinhong Hei, Zongjian Li, "Semiautomatic Generation of Code Ontology Using ifcOWL in Compliance Checking", *Advances in Civil Engineering*, vol. 2021, Article ID 8861625, 18 pages, 2021. https://doi.org/10.1155/2021/8861625

[3] Junjun Guo, Haonan Li, Zhengyuan Wang, Li Zhang, Changyuan Wang, "A Novel Vulnerable Code Clone Detector Based on Context Enhancement and Patch Validation", *Wireless Communications and Mobile Computing*, vol. 2022, Article ID 3822836, 12 pages, 2022. https://doi.org/10.1155/2022/3822836

[4] Shobha G, Rana A, Kansal V, Tanwar S. Code Clone Detection—A Systematic Review. In: Hassanien AE, Bhattacharyya S, Chakrabati S, Bhattacharya A,editors. Emerging Technologies in Data Mining and Information Security;vol. 1300. Springer. 2021;p. 645–655. Available from: https://doi.org/10.1007/978-981-33-4367-2_61.

[5] Sachin Lalar, Shashi Bhushan, Surender Jangra, Mehedi Masud, Jehad F. Al-Amri, "An Efficient Three-Phase Fuzzy Logic Clone Node Detection Model", *Security and Communication Networks*, vol. 2021, Article ID 9924478, 17 pages, 2021. https://doi.org/10.1155/2021/9924478

[6] Lan Zhang, Hao Hu, Yi Fang, ZhenyuQiang, "Code Compliance in Reinforce Concrete Design: A Comparative Study of USA Code (ACI) and Chinese Code (GB)", *Advances in Civil Engineering*, vol. 2021, Article ID 5517332, 9 pages, 2021. https://doi.org/10.1155/2021/5517332

[7] ZhihuaZha, Chaoqun Li, Jing Xiao, Yao Zhang, Hu Qin, Yang Liu, Jie Zhou, Jie Wu, "An Improved Adaptive Clone Genetic Algorithm for Task Allocation Optimization in ITWSNs", *Journal of Sensors*, vol. 2021, Article ID 5582646, 12 pages, 2021. https://doi.org/10.1155/2021/5582646

[8] Xingzheng Li, Bingwen Feng, Guofeng Li, Tong Li, Mingjin He, "A Vulnerability Detection System Based on Fusion of Assembly Code and Source Code", *Security and Communication Networks*, vol. 2021, Article ID 9997641, 11 pages, 2021. https://doi.org/10.1155/2021/9997641

[9] Harlinah Sahib, WaodeHanafiah, Muhammad Aswad, Abdul Hakim Yassi, Farzad Mashhadi, "Syntactic Configuration of Code-Switching between Indonesian and English: Another Perspective on Code-Switching Phenomena", *Education Research International*, vol. 2021, Article ID 3402485, 10 pages, 2021. https://doi.org/10.1155/2021/3402485

[10] Yao Zhang, Yan Liu, Chaoqun Li, Yang Liu, Jie Zhou, "The Optimization of Path Planning for Express Delivery Based on Clone Adaptive Ant Colony Optimization", *Journal of Advanced Transportation*, vol. 2022, Article ID 4825018, 15 pages, 2022. https://doi.org/10.1155/2022/4825018

[11] Rui Yang, Mengying Xu, Jie Zhou, "Clone Chaotic Parallel Evolutionary Algorithm for Low-Energy Clustering in High-Density Wireless Sensor Networks", *Scientific Programming*, vol. 2021, Article ID 6630322, 13 pages, 2021. https://doi.org/10.1155/2021/6630322

[12] Guohong Qi, Jie Zhou, Wenxian Jia, Menghan Liu, Shengnan Zhang, Mengying Xu, "Intrusion Detection for Network Based on Elite Clone Artificial Bee Colony and Back Propagation Neural Network", *Wireless Communications and Mobile Computing*, vol. 2021, Article ID 9956371, 11 pages, 2021. https://doi.org/10.1155/2021/9956371

[13] Yanhong Qi, Li-Ping Wang, "A New Code-Based Traceable Ring Signature Scheme", *Security and Communication Networks*, vol. 2022, Article ID 3938321, 10 pages, 2022. https://doi.org/10.1155/2022/3938321

[14] Jing Xiao, Yang Liu, Hu Qin, Chaoqun Li, Jie Zhou, "A Novel QoS Routing Energy Consumption Optimization Method Based on Clone Adaptive Whale Optimization Algorithm in IWSNs", *Journal of Sensors*, vol. 2021, Article ID 5579252, 14 pages, 2021. https://doi.org/10.1155/2021/5579252

[15] Yang Li, Fei Kang, Hui Shu, Xiaobing Xiong, Zihan Sha, Zhonghang Sui, "COOPS: A Code Obfuscation Method Based on Obscuring Program Semantics", *Security and Communication Networks*, vol. 2022, Article ID 6903370, 15 pages, 2022. https://doi.org/10.1155/2022/6903370

[16] Jia Chen, Quankai Qi, Yongjie Wang, Xuehu Yan, Longlong Li, "Data Hiding Based on Mini Program Code", *Security and Communication Networks*, vol. 2021, Article ID 5546344, 12 pages, 2021. https://doi.org/10.1155/2021/5546344

[17] Ke Tang, Zheng Shan, Fudong Liu, Yizhao Huang, Rongbo Sun, Meng Qiao, Chunyan Zhang, Jue Wang, Hairen Gui, "SROBR: Semantic Representation of Obfuscation-Resilient Binary Code", *Wireless Communications and Mobile Computing*, vol. 2022, Article ID 4095481, 11 pages, 2022. https://doi.org/10.1155/2022/4095481

[18] Yao Meng, "An Intelligent Code Search Approach Using Hybrid Encoders", *Wireless Communications and Mobile Computing*, vol. 2021, Article ID 9990988, 16 pages, 2021. https://doi.org/10.1155/2021/9990988

[19] Massimiliano Ferraioli, "Behaviour Factor of Ductile Code-Designed Reinforced Concrete Frames", *Advances in Civil Engineering*, vol. 2021, Article ID 6666687, 18 pages, 2021. https://doi.org/10.1155/2021/6666687

[20] Yang Liu, Jing Xiao, Chaoqun Li, Hu Qin, Jie Zhou, "Sensor Duty Cycle for Prolonging Network Lifetime Using Quantum Clone Grey Wolf Optimization Algorithm in Industrial Wireless Sensor Networks", *Journal of Sensors*, vol. 2021, Article ID 5511745, 13 pages, 2021. https://doi.org/10.1155/2021/5511745

[21] Wanzhi Wen, Jiawei Chu, Tian Zhao, Ruinian Zhang, Bao Zhi, Chenqiang Shen, "Code2tree: A Method for Automatically Generating Code Comments", *Scientific Programming*, vol. 2022, Article ID 6350686, 9 pages, 2022. https://doi.org/10.1155/2022/6350686

[22] Zixuan Rui, Xiaofeng Ouyang, Fangling Zeng, Xu Xu, "Blind Estimation of GPS M-Code Signals under Noncooperative Conditions", *Wireless Communications and Mobile Computing*, vol. 2022, Article ID 6597297, 14 pages, 2022. https://doi.org/10.1155/2022/6597297

[23] M. Hammad, Ö. Babur, H. A. Basit and M. Van Den Brand, "Clone-Seeker: Effective Code Clone Search Using Annotations," in IEEE Access, vol. 10, pp. 11696-11713, 2022, doi: 10.1109/ACCESS.2022.3145686.

[24] W. Hua, Y. Sui, Y. Wan, G. Liu and G. Xu, "FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks," in IEEE Transactions on Reliability, vol. 70, no. 1, pp. 304-318, March 2021, doi: 10.1109/TR.2020.3001918.

[25] F. Zhang, S. -C. Khoo and X. Su, "Improving Maintenance-Consistency Prediction During Code Clone Creation," in IEEE Access, vol. 8, pp. 82085-82099, 2020, doi: 10.1109/ACCESS.2020.2990645.

[26] A. Sheneamer, S. Roy and J. Kalita, "An Effective Semantic Code Clone Detection Framework Using Pairwise Feature Fusion," in IEEE Access, vol. 9, pp. 84828-84844, 2021, doi: 10.1109/ACCESS.2021.3079156.

[27] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu and C. K. Roy, "LVMapper: A Large-Variance Clone Detector Using Sequencing Alignment Approach," in IEEE Access, vol. 8, pp. 27986-27997, 2020, doi: 10.1109/ACCESS.2020.2971545.

[28] H. Zhang and K. Sakurai, "A Survey of Software Clone Detection From Security Perspective," in IEEE Access, vol. 9, pp. 48157-48173, 2021, doi: 10.1109/ACCESS.2021.3065872.

[29] J. Svajlenko and C. K. Roy, "The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis," in IEEE Transactions on Software Engineering, vol. 47, no. 5, pp. 1060-1087, 1 May 2021, doi: 10.1109/TSE.2019.2912962.

[30] C. Guo et al., "Review Sharing via Deep Semi-Supervised Code Clone Detection," in IEEE Access, vol. 8, pp. 24948-24965, 2020, doi: 10.1109/ACCESS.2020.2966532.

[31] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue and Y. Xu, "Challenging Machine Learning-Based Clone Detectors via Semantic-Preserving Code Transformations," in IEEE Transactions on Software Engineering, vol. 49, no. 5, pp. 3052-3070, 1 May 2023, doi: 10.1109/TSE.2023.3240118.

[32] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco and R. Oliveto, "Toxic Code Snippets on Stack Overflow," in IEEE Transactions on Software Engineering, vol. 47, no. 3, pp. 560-581, 1 March 2021, doi: 10.1109/TSE.2019.2900307.

[33] Z. Li, T. -H. Chen, J. Yang and W. Shang, "Studying Duplicate Logging Statements and Their Relationships With Code Clones," in IEEE Transactions on Software Engineering, vol. 48, no. 7, pp. 2476-2494, 1 July 2022, doi: 10.1109/TSE.2021.3060918.

[34] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo and R. Purandare, "Modeling Functional Similarity in Source Code With Graph-Based Siamese Networks," in IEEE Transactions on Software Engineering, vol. 48, no. 10, pp. 3771-3789, 1 Oct. 2022, doi: 10.1109/TSE.2021.3105556.

[35] F. Khan, I. David, D. Varro and S. McIntosh, "Code Cloning in Smart Contracts on the Ethereum Platform: An Extended Replication Study," in IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 2006-2019, 1 April 2023, doi: 10.1109/TSE.2022.3207428.

[36] Y. Hu, G. Xu, B. Zhang, K. Lai, G. Xu and M. Zhang, "Robust App Clone Detection Based on Similarity of UI Structure," in IEEE Access, vol. 8, pp. 77142-77155, 2020, doi: 10.1109/ACCESS.2020.2988400.

[37] H. Koo, S. Park, D. Choi and T. Kim, "Binary Code Representation With Well-Balanced Instruction Normalization," in IEEE Access, vol. 11, pp. 29183-29198, 2023, doi: 10.1109/ACCESS.2023.3259481.

[38] Y. Hu, H. Wang, Y. Zhang, B. Li and D. Gu, "A Semantics-Based Hybrid Approach on Binary Code Similarity Comparison," in IEEE Transactions on Software Engineering, vol. 47, no. 6, pp. 1241-1258, 1 June 2021, doi: 10.1109/TSE.2019.2918326.

[39] A. M. Sheneamer, "An Automatic Advisor for Refactoring Software Clones Based on Machine Learning," in IEEE Access, vol. 8, pp. 124978-124988, 2020, doi: 10.1109/ACCESS.2020.3006178.

[40] N. -T. Chau and S. Jung, "Enhancing Notation-Based Code Cloning Method With an External-Based Identifier Model," in IEEE Access, vol. 8, pp. 162989-162998, 2020, doi: 10.1109/ACCESS.2020.3016943.

[41] Z. Gao, L. Jiang, X. Xia, D. Lo and J. Grundy, "Checking Smart Contracts With Structural Code Embedding," in IEEE Transactions on Software Engineering, vol. 47, no. 12, pp. 2874-2891, 1 Dec. 2021, doi: 10.1109/TSE.2020.2971482.

[42] H. Wang et al., "Enhancing DNN-Based Binary Code Function Search With Low-Cost Equivalence Checking," in IEEE Transactions on Software Engineering, vol. 49, no. 1, pp. 226-250, 1 Jan. 2023, doi: 10.1109/TSE.2022.3149240.

[43] V. Sharma, K. Hietala and S. McCamant, "Finding Substitutable Binary Code By Synthesizing Adapters," in IEEE Transactions on Software Engineering, vol. 47, no. 8, pp. 1626-1643, 1 Aug. 2021, doi: 10.1109/TSE.2019.2931000.

[44] D. Pizzolotto and K. Inoue, "BinCC: Scalable Function Similarity Detection in Multiple Cross-Architectural Binaries," in IEEE Access, vol. 10, pp. 124491-124506, 2022, doi: 10.1109/ACCESS.2022.3225100.

[45] I. Reinhartz-Berger and A. Zamansky, "Reuse of Similarly Behaving Software Through Polymorphism-Inspired Variability Mechanisms," in IEEE Transactions on Software Engineering, vol. 48, no. 3, pp. 773-785, 1 March 2022, doi: 10.1109/TSE.2020.3001512.

[46] Kumar, M., et al. "Self-attentive CNN+ BERT: An approach for analysis of sentiment on movie reviews using word embedding." Int J Intell Syst Appl Eng 12 (2024): 612.

[47] Narayan, Vipul, et al. "A comparison between nonlinear mapping and high-resolution image." Computational Intelligence in the Industry 4.0. CRC Press, 2024. 153-160.

[48] J. Gao et al., "Semantic Learning and Emulation Based Cross-Platform Binary Vulnerability Seeker," in IEEE Transactions on Software Engineering, vol. 47, no. 11, pp. 2575-2589, 1 Nov. 2021, doi: 10.1109/TSE.2019.2956932.

[49] Y. Zhao, L. Xiao, A. B. Bondi, B. Chen and Y. Liu, "A Large-Scale Empirical Study of Real-Life Performance Issues in Open Source Projects," in IEEE Transactions on Software Engineering, vol. 49, no. 2, pp. 924-946, 1 Feb. 2023, doi: 10.1109/TSE.2022.3167628.

[50] H. Hong, S. Woo, E. Choi, J. Choi and H. Lee, "xVDB: A High-Coverage Approach for Constructing a Vulnerability Database," in IEEE Access, vol. 10, pp. 85050-85063, 2022, doi: 10.1109/ACCESS.2022.3197786.