Received: July 26, 2022. Revised: September 13, 2022.

International Journal of
Intelligent Engineering & Systems

http://www.inass.org/

A Collaborative method for Code Clone Detection Using Lexical, Syntactic, Semantic and Structural Features

Karthik Sekar[1*] **Rajdeepa Balasubramanian**[2]

[1]*Department of Computer Science, PSG College of Arts and Science (Autonomous), Coimbatore, Tamilnadu, India*
[2]*Department of Information Technology, PSG College of Arts and Science (Autonomous) Coimbatore, Tamilnadu, India*
* Corresponding author's Email: karthikyessekarphd@gmail.com

Abstract: Software code clones (CC) in software programs are degrading the performance of software systems. many code clones detection (CCD) methods proposed in the literature detect only individual cloned types efficiently. This paper proposes a collaborative code clones detection (CCCD) method by utilizing lexical, syntactic, semantic and structural features for effectively identifying all types of clones including type-4. Initially, a large variance mapper (LV-mapper) is utilized to identify clone pairs (CPs). Then CPs are converted into lexical features by directly applying Word2vec. The synonyms of CPs are obtained using the WordNet tool and converted as semantic features by Word2vec.Additionally, code size metrics (CZM) and object oriented metrics (OOM) are additionally measured as structural features of a program's code blocks (CBs). The syntactic features are extracted through the abstract syntax tree (AST) from the source code. Finally, the joint feature vector is generated by combining all the features together. In order to detect CCs in any new software, the joint feature vector of known clone type source codes is generated first (training set) and then the joint feature vector of unknown clone types source codes is generated next. The Euclidean distances between training and testing of joint feature vectors determine the clone type of test features. Finally, the experimental outcome demonstrates that the proposed CCCD technique has an accuracy of 87.8%, 92.3% and 95.5% for the dataset Apache Maven 3.8.3, Appache ant 1.10.12 and Opennlp-master 1.9.1, respectively, compared to the existing LV-CCD, ES-CCD, TBCNN-CCD and CPVDetector methods.

Keywords: Code clone detection, LV-mapper, Clone pairs, Joint feature vector, Euclidean distances.

1. Introduction

Cloned code, abbreviated as CC, is common in software development and is created by duplicating a section of code with little or no changes into another section of code. Because of CCs, errors available in a single section of the program affected all duplicated sections. As a result, identifying CCs in all segments inside the source code is essential. According to several assessments, CCs available in 20-50% of large code size software projects [1, 2].

CCs harm many software engineering functions, such as aspect mining, program comprehension, software assessment, program code efficiency evaluation, bug and virus identification. CCs also induce bug propagation which greatly raises the cost of software maintenance. Because of these upkeep issues, CCD has emerged as a hot topic in research. The types of CCs are as follows

- **Type-1** clones (Lexical similarity): This type of clone is substantially identical except for variations in variable names, function names, white space, formatting, and comments.
- **Type-2** clones (Semantic similarity): These clone types can be identified by different code snippets that implement the same behaviour in a syntactic manner.
- **Type-3** clones (Structural similarity): These CCs contain comparable software structures (for example, design patterns and object oriented programming class interactions). This

resemblance extends to the syntactic and logical study of code structure in programs.

- **Type-4** clones (Syntactic similarity): This can be found in code snippets that are similar at the statement level but differ at the code level. Statements have been added, altered, or removed in code samples.

Text, Token, Tree, metric, Semantic, and Hybrid based methods are commonly used for CCD [3, 4]. Various tools for CCD are NICAD, CCFinderX, Simian, and CPMiner [5, 6]. Various similarity measures like Fingerprinting, Neural networks and Euclidian distance are used to detect CCs [7]. However, because most of the available tools and methodologies were designed for highly similar clones, they are incapable of detecting large-variance CCs (LV-CC). These high-variability clones can be identified in a variety of software applications.LV-CC detection is more important for software maintenance and malware detection.

CCAligner [8] discovered LV-CCs effectively. The duplication or alteration in a single spot is known as "gap", and this gap in clones typically results in an LV-CCs. LV-CCs propagation is effectively found across several locations rather than just one. However, this method results in limited scalability, also performs well only for type1 and type2 CCs.

A unique and effective detector called LV-Mapper [9] known as LV-CCD uses a locate-filter-verify technique which locates and filters probable clone code by using a restricted window of continuous lines known as seeds which determines the dynamic threshold for CC verification. It detects Type-1 to Type-3 clones more accurately. However, it is incapable of producing satisfactory results for type-4 clones.

In this research work, a CCCD approach is proposed to recognize all sorts of clones (Type 1 to Type 4) successfully. The proposed CCCD approach makes use of various features such as lexical, syntactic, semantic, and structural. This approach tokenizes all CPs detected by LV-Mapper and utilizes the WordNet tool to search for synonyms for tokens. Word2vec is used to translate synonyms into semantic features. CBs, CZM and OOM are measured for structural features. AST is used to capture syntactic features from the source code. The lexical, syntactic, semantic, and structural features are then combined to form a joint feature vector. Euclidean distances between the training joint feature vector and testing joint feature vector compute all clone types with less computational challenges.

The next section of this paper explains existing CCD approaches. Section 3 explains the proposed methodology. Section 4 briefly describes the outcome of the evaluation. Section 5 explains conclusion and future scope.

2. Literature survey

A token-based CCD with an adaptive partitioning method was developed to detect Type-3 clones [10]. The volume of code segments was assessed in filtering stage. Non-potential segments for CCs are removed dramatically to decrease complexity. The validation stage confirmed the candidate pairs that are actual CCs. This two-stride method shortens the CCD runtime. However, this method has a higher chance of failing in the detection process while testing bigger inputs.

A test-based approach was developed [11] for detecting semantic clones. This approach was applied to detect semantic clones of API methods. The test cases for a given method were generated automatically, and the generated test cases were used to search for semantically equivalent API methods. When two methods produce the same output on all of these test cases, they are termed semantically equivalent methods. This approach was restricted to arbitrary chunks to reduce time complexity. However, the appropriate selection of a test case generation tool is an issue for this method.

A Modular, Sequential, and Multi-representation clone search engine called Siamese was proposed for CC search, [12]. This approach converts Java code into four program connections for the simultaneous identification of numerous clone varieties. A query reduction (QR) approach was utilized to reduce query size based on token document frequencies. Incremental index updating, allows for quick updates from existing indexes without having to construct the index from scratch. The siamese clone search technique enables real-time discovery of online code reuse, comparable code samples and software plagiarism detection. However, these evaluated methodologies were too small to be indicative of the software sector.

An efficient semantic CCD (ES-CCD) technique was developed [13] by employing a pair-wise feature fusion for automated identification of all four clone types. AST and program dependency graphs (PDGs) were efficiently utilized to prepare labelled training features for detecting the Java code clones, including semantic clones. Then, the full path traversal method was used to extract the AST and PDG features and helps to derive those features in vector formation. The syntax of program codes was captured using AST program features, and the semantics of program codes using PDG features. Then the machine learning method was employed to find the relative

performance of the model using these extracted features. However, this model was acquired at a high computational cost.

Tree-based convolution neural network (TBCNN) was developed [14] for CCD. This method utilizes two-pass technique to detect and classify the clones-types based on their code features. These features were readily captured by using the AST node that reflects typical code patterns which was created from code and saves time on the preparation stage. This process uses two classification models like (i) 1-st pass classification would detect the clone based on the provided clone fragments. (ii) 2nd-pass classification would classify the clone code type. However, this model was insufficient to handle the fine-grained code fragments.

A CCD technique was presented [15] using code fingerprints, known as the context-enhanced and patch-validation-based vulnerable code clone detector (CPVDetector). In this technique, a fingerprint database was created for functions, code fragments, and patches that were obtained from pre-processed susceptible source code. First, the target code that needs to be detected was converted into function-level fingerprints. If this coarse granularity could not detect clones, the detector could be used to detect them. The detector would move to check the context of vulnerable codes after a successful fingerprint match between the target code and the vulnerable code segments. However, this model has lower performance on larger datasets.

2.1 Research contribution

From the above literature survey, it is clear that most of the existing methods that have been developed for CCD are highly suitable to detect general Type-1 to Type-3 clones. But not having sufficient proof to provide better results for type-4 clones. So, the proposed work is developed using lexical, syntactic, semantic and structure based features which are combined together to detect all four clone types (Type 1 to Type 4). The source code data is taken from the open source application which is split into training and testing data. In this research work, four primary clone types and their similarity features are used in this process. The further processes of this research work are briefly explained below.

3. Proposed methodology

3.1 Extraction of CBs

A program unit is a series of statements surrounded by braces that normally represent a single function. This CB is crucial in detecting CPs with identical code sections. Lexical analysis for code in this system entails extracting CBs from source code and transforming them to Turing eXtender Language (TXL) [16]. TXL is a coding language developed for modifying program coding transcriptions and attributes via link transformation. The Fig. 1 depicts the reliable description of the proposed CCCD model.

TXL's guiding principle is to start with a syntax for an existent system, define syntactic changes to the grammar that reflect new language attributes or expansions, and then quickly test-type these novel attributes through code conversion to the source language. It extracts CBs by specifying syntactical rules in terms of hash tags (restrained signifiers), substances (several integrity succession to be treated as a unit), remarks (definition of articulating norms), and more broadly, tokens\symbols, structured pattern matching for random character concatenation. These sentences will be useful in determining the structure of CBs.

3.2 Tokenizing the CBs

After extracting the CBs, the tokenizing step is carried out using a tool called fast lexical analyzer generator (Flex). Flex tool [17] is used for the tokenizing these extracted CBs. During the process, the extracted CBs are used as input files, which are generated by a lexical analyzer and may be recast in lex language. The lex compiler transforms a input file to specific code program (C program). Then, the specific code program will compile the input file into an executable file or output file. From this output file, a stream of input characters is analyzed and produces a stream of token codes from the extracted CBs which is then indexed to detect the clone pairs.

3.3 Detecting CPs

LV-Mapper [9] examines all operational modules and accumulates all seeds instead of tokens to discover clone pairs. These seeds are transformed into hash values and saved in a hash table. The seed’s hash is the key of the component in the hash table in this process, and the content is an accumulation of related unit’s ids.

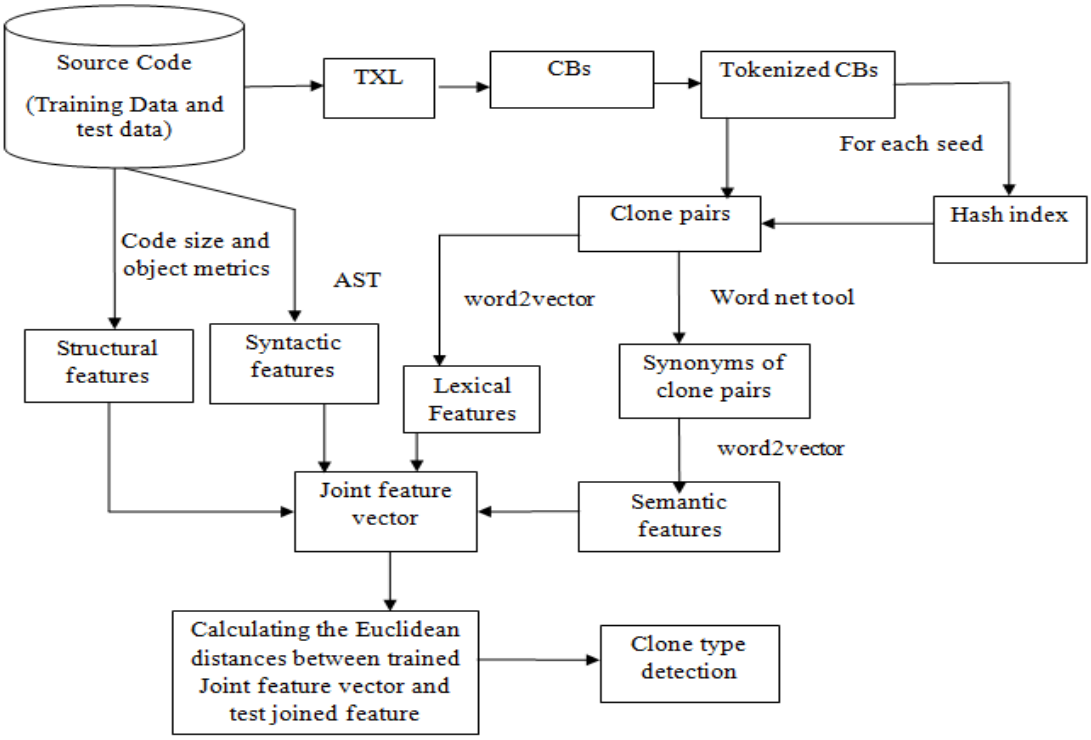

Figure. 1 Systematic representation of the developed CCCD method

3.3.1 Identifying prospective CC pairs using the shared seed:

It is done by searching for probable CC pairings using the indexed seed. The purpose of this phase is to collect as many candidates as possible while minimising the loss of actual clone couplings. In this strategy, the 3-line sliding frames are employed as the seeds region to combine all potential clone sets that contain the similar seed (s) and allow improved clone pair determination.

3.3.2 Retrieval for the quantity of common seeds:

The possibility of CBs that might be easily duplicated is taken into account. This is the filtering step, and it is in charge of identifying potential clone pairings. Filtering step is especially responsible with estimating the possibility of cloning by assessing the quantity of common seeds between two CBs. The resemblance of the two CBs, as measured by the number of seeds they possess, determines the choosing of probable CPs. In this method, the resemblance (S_r) of code pairs A and B is computed as

$$S_r(B\backslash A) = \frac{s}{t} = \frac{s}{L-k+1} \quad (1)$$

The amount of shared seeds is s, the overall number of seeds in code pair B is t, and the length of B in line is L. The greater the $S_r(B\backslash A)$ value for each pair of CBs, the more likely it is that they are clone pairs.

From the following procedures, the CPs are detected. Once the clone pair is identified, similarity features such as lexical and semantic similarity must be retrieved from these pairs using various tools for detecting clone types as detailed in detail below whereas other two similarity features (structural and syntactic) are extracted directly from the source code.

3.4 Lexical feature

Whole word, prefix/suffix (different lengthsallowed), stemmed word, lemmatized word are the most common lexical features. To detect the CC from the clone pairs, this tool will compute the similarity of hamming space instead of vector pairs created by binary hash functions. For the extraction of lexical features from the clone pairs, the word embedding technique like word2vector (word2vec) tool [18] is used to capture the word context in a data, semantic and syntactic similarity, relation with other words, etc.

To detect the CC from the CPs, this tool will compute the similarity of hamming space instead of vector pairs created by binary hash functions. It parses feature text by "vectorizing" words using a

two-strand network. It keeps a textual sample as input to produce a set of arrays: attribute matrix describing a words in the article. This Word2vec tool is widely used in natural language processing (NLP) to shorten learning time. Word2Vec may use two distinct model architectures to generate these word embedding representations like the continuous bag of words (CBOW) model and the skip-gram model. CBOW is used in this framework because it is quicker and provides better representations for more common and related terms, even in greater datasets.

3.4.1 Structure of CBOW model

This model efficiently attempts to predict the lexical feature based on the identified clone pairs' source code words. Also, the model converts the sentences into word pairs in the form ($source\_codeword, target\_word$). With these word pairs, the model attempts to predict the lexical feature based on the source code words using the user-adjusted window size. The Fig. 2 depicts the CBOW model structure. In the figure, the four source code words ($w_{t-2}, w_{t-1}, \cdots w_{t+1} and\, w_{t+2}$) has been used for the prediction of lexical features. The input will be in the form of $1 \times W$ input vectors. These input vectors are sent to the buried layer, where they are multiplied by a WXN matrix. Finally, the $1XN$ output from the hidden layer enters the sum layer, where the vectors are element-wise summed before a final activation is done and the output is generated.

The CBOW approach is constructed from a neural network language model (NNLM) that simultaneously develops a phrase embedding and a language concept, with the exception that there is no quadratic. The CBOW model is designed to forecast the median phrase given $N/2$ past chronological terms and $N/2$ prospective phrases. When $N = 8$ is employed, the best findings are produced. Word arrays of N input sentence are simply summed in the interpreted stage. The positioning of the word has minimal effect on determining the word vector of the middle word, hence the term "Bag of words.

The latent factor D is represented by the word Continuous. The CBOW model is a simple log-linear paradigm in which the logarithmic of the model's outcome may be written as a quadratic combination of the model's parameters. To get distribution over V, an average variable is sent to the out-layer, which is then accompanied by the recursive softmax (V is the total number of words in the corpus). The overall weights used in developing the CBOW system are $N \times D + D \times log(2)V$.

The CBOW variables are two groups of word extracted features: "reference-edge" and "desired-edge" vectors v_w, $v'_w \in R^d$ for each, V lexicon word type $w \in V$. A text window in a corpus consists of a central word w o and context words $w_{1,\ldots}w_c$. For example, *the dog laughed in the window*, $w_o = dog$, $w_1 = the$, $w_1 = laughed$. Given a text window, the CBOW loss is defined as:

$$v_c = \frac{1}{c}\sum_{j=1}^{c} vw_j$$
$$\mathcal{L} = -log\,\sigma\left(v'_{wO}{}^{\mathcal{T}v_c}\right) - \sum_{i=1}^{k} log\,\sigma\left(v'_{n_i}{}^{\mathcal{T}v_c}\right) \quad (2)$$

Where $n_1 \ldots . n_k \in V$ are the negative samples of a noise distribution P_n over V . $\mathcal{L}$ gradients with reference to the desired source (v'_{wO}),), negotiate source (v'_{n_i}) , and standard topic origin v_c embeddings are produced.

$$\frac{\partial L}{\partial v'_{wO}} = \left(\sigma\left(v'_{wO}{}^{\mathcal{T}v_c}\right) - 1\right)v_c$$
$$\frac{\partial L}{\partial v'_{n_i}} = \left(\sigma\left(v'_{n_i}{}^{\mathcal{T}v_c}\right)v_c\right.$$
$$\frac{\partial L}{\partial v_c} = \left(\sigma\left(v'_{wO}{}^{\mathcal{T}v_c}\right) - 1\right)v'_{wO} + \sum_{i=1}^{k} = (\sigma\left(v'_{n_i}{}^{\mathcal{T}v_c}\right)v'_{n_i} \quad (3)$$

Hence, in the case of a source context embedding, by the chain rule:

$$\frac{\partial L}{\partial v_{w_j}} = \frac{1}{c}\left[\left(\sigma\left(v'_{wO}{}^{\mathcal{T}v_c}\right) - 1\right)v'_{wO} + \sum_{i=1}^{k} = (\sigma\left(v'_{n_i}{}^{\mathcal{T}v_c}\right)v'_{n_i}\right. \quad (4)$$

In this method, the building-method will be used to tokenize every feature in the source context and try to fit the word in the tokenizer. The total number of features will be calculated for further use. The window size is then determined by the greatest distance between the target words (lexical characteristics) and their contextually adjacent words. Then, a function is created to match the context and target terms. The function that was made takes the widths of the target window and the context window separately and makes pairs of contextual words and target words (lexical features).

3.5 Semantic feature

The word net tool comprises a lexical database of words in over 200 languages, with adjectives,

adverbs, nouns, and verbs arranged separately into a series of conceptual analogues, and all phrases in the database will expresses a separate conception. Intellectual analogues, also known as synsets, are provided by utilizing conceptual-semantic and lexical linkages. WordNet looks to function similarly to a lexicon, by clustering terms collectively depending on their definitions. Nonetheless, there are numerous distinctive traits.

- Initially, WordNet integrates not only phrase structures and letter strings, but also actual word meanings. As a result, the channel's contextually linked terms are meaningfully interpreted.
- Second, WordNet recognizes meaningful links between phrases, whereas thesaurus word classifying does not predefined structure other than meaning similarity.

Algorithm for the extraction of tokens

Required $= d\{w_i, \ldots., w_n\}$: $w_i \in$ $lexical\ database\ (Wordnet)$
Require: $ld = lexical\ database$
function Source(d, ld) \\ d - document containing words w_n, ld - lexical data base
2. $list\_of\_tokens = \oslash$
3. $\boldsymbol{For}\ i = 0\ to\ n\ \ \boldsymbol{do}$
4. $present = synsets(w_i, ld)$
5. $If\ i \neq 0 \wedge i \neq n\ then$
6. $previous = synsets(w_{i-1}, ld)$
7. $next = synsets(w_{i+1}, ld)$
8. $\boldsymbol{else\ if}\ i = 0\ \boldsymbol{then}$
9. $previous = \oslash$
10. $next = synsets(w_{i+1}, ld)$\
11. $\boldsymbol{else}$
12. $previous = synsets(w_{i-1}, ld)$
13. $next = \oslash$
14. $present\_candidtaes = \oslash,\ previous\_candidates = \oslash, and\ next\_candidates = \oslash$
15. $\boldsymbol{for}\ s_{pr} \in \{present\}, s_{pi} \in \{previous\}\ and\ s_n \in \{next\}\ do$ \\where $0 \leq c \leq Q, 0 \leq f \leq R$ and $0 \leq l$
16. $\boldsymbol{Insert}\ synset - vec(s_{pr}, s_{pi}, s_n)\ \boldsymbol{to}\ present\_candidates, previous\_candidates, next\_candidates$
17. $u = argmax_{s_{ca}} \{cosine - similarity(present\_candidates, previous\_candidates)\}$
18. $w = argmax_{s_{cb}} \{cosine - similarity(present\_candidates, next\_candidates)\}$
19. $\boldsymbol{Insert}\ synset\ (s_{ca}\ or\ s_{cb})$ $with\ the\ highest\ produced\ cosine\ similarity$ $\boldsymbol{to}\ list\_of\_tokens$
20. $Return\ list\_of\_tokens$

The $list\_of\_tokens$ from the above algorithm is considered as synonyms.

Finally, the produced synonyms are converted as synset-based vector representations which are then used as an input in a word2vec with CBOW as the training algorithm for the extraction of semantic features between synsets.

3.6 Structural feature

The rational and syntactic investigation of source scripts is part of clone structural analysis. The software's layout patterns are utilized in the evaluation of structural clones as a representation of rational assessment. CZM and OOM [19] are used to indicate the attributes in source code for combining structural similarities of Functionality.

(A) CZM: When the programs of the items are exactly duplicated and exploited without modifications, the dimension parameters may be utilised to detect clones.

The following names have been used for this CZM:

(i) Line score: The entire volume of lines in the code program

(ii) Total integer of comments: The maximum integer of comments in the source code.

(iii) Lines of code commented and uncommented: The overall amount of commented and uncommented LOC in the source code.

(iv) Token count: The total score of tokenized Lines Of Code (LOC) in the source code.

(v) Addition of entire identifier: The value of tokenized identifiers in the program as a whole

(B) OOM: Several key aspects of an object-oriented language are required to construct object-oriented software. Abstraction, polymorphism, and inheritance are examples of these characteristics. The inclusion of all components required for an entity to perform successfully, most notably methods and data, is referred to as abstraction. Any sub-type that is formed after an object class is declared may acquire the definitions of one or multiple universal labels. The various names used for object-oriented metrics are as follows:

(i) Depth in the tree: A class's number of inheritance tiers

Purpose: Same depth value of classes in the project that are architecturally comparable to each other.

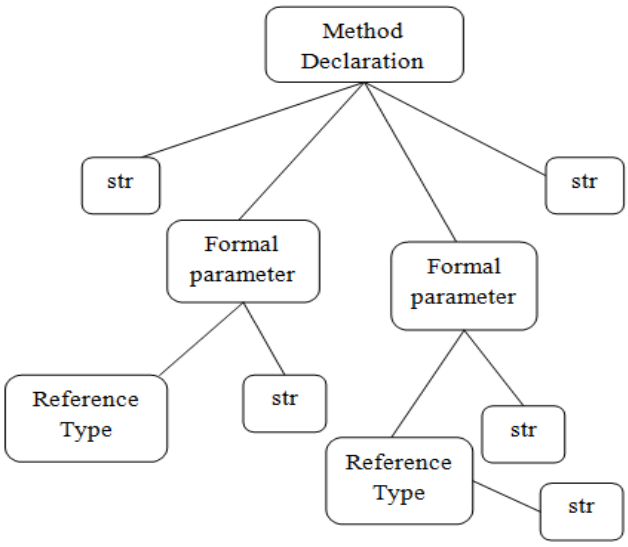

Figure. 2 Sample AST with generate features vector

(ii) Number of users who use another class: The total number of additional classes utilised in the class.

Purpose: The same number class value is utilised in all classes in the program, even if they are substantially identical.

(iii) Total number of children: The overall number child in the class.

Purpose: The goal is to have the same amount of child values of classes in the program that are architecturally comparable to each other.

(iv) Number of other classes that use: Total amount of class times used by others

Proposal: Use the identical employed occurrences values of labels by other conceptually relevant objects in the application.

(v) Amount of arguments: The total number of parameters supplied to the method

Proposal: The goal is to have the same argument number value of methods in the project that are architecturally comparable to each other.

(vi) The number of parameters returned Method returns the total number of parameters.

Purpose: The objective is for all methods in the project that are structurally equivalent to have the same returning argument number value.

(vii) The number of times you've called other methods: The total number of calling methods in a single method.

Purpose: Use the same amount of calls to other method values in the project, which might have comparable structures.

This statistic is used to determine which grades are functionally related to each other by quantifying the proximity among observation variables. A linear array of evaluated results is represented by each source code that is being evaluated for each of its labels. If the cosine and jaccard functions [19] both yield normalized resemblance findings towards a certain pair of classes that are within the set threshold values, these classes are retained as structural features.

3.7 Syntactic feature

An AST is a tree representation of a code fragment. To extract the syntactic features from the code fragment, code fragments is fragmented into different parts, then the code is converted into a set of tokens, and the list of tokens is turned into an AST [20]. Each node of the AST tree structure has a type specifying what it is representing. For example, a type could be a "MethodDeclaration" representing a method definition or a "FormalParameter" representing a parameter for a method declaration. There are two "FormalParameter" subtrees, each with a "ReferenceType" of "str", that is, String.

3.7.1 Vector representation of AST

To facilitate data mining on code and an interpretation of the data mining results, syntax trees should be transformed into continuous vectors for representing the code. Vector representation of the code enables a much more comprehensive range of analysis. Since machine learning algorithms take vectors as their inputs, the vector embedding techniques [21] is used to transform AST into vectors. The code vectors capture properties of code fragments, such that similar code fragments have similar vectors. Fig. 2 represents the sample AST with generates feature vectors.

Type your main text in 11-point Times New Roman, single-spaced. Do not use double-spacing. All paragraphs should be indented 1.5 times character size. Be sure your text if fully justified—that is, flush left and flush right. Please do not place any additional blank lines between paragraphs.

The AST represented for syntactic feature extraction [22] is used in this work, which is directly applied to the source code for extracting syntactic features of functionality. For example, C is a code snippet, and N_{root} is the AST entry node that corresponds to it. The beginning section of N_{root} is iterated over all the nodes of AST in preorder to extract the syntactic representation of C. There is an identifier, such as symbols and variable names, in each AST node. The identification sequence $Seq = \{ident1, ident2, \ldots, identn\}$ is formed, and it may be used to express C's syntactic information.

Each method in source code is now represented as a sequence of identifiers. Average pooling used in

[22] is applied to all identifier vectors in each procedure to provide a syntactic feature vector. Following that, each method's syntactic information is encoded as a fixed-length feature vector marked a

$$mv = average(h_i), i = 1, \ldots, N \quad (5)$$

Where h_i is the identifier's feature vector, Average pooling method provide the relevant syntactic feature for each functionality (i.e., the related methods). Each functionality's syntactic information is expressed as a fixed-length feature vector

$$fv = average(mv_i), i = 1, \ldots, N \quad (6)$$

Where mv_i is the syntactic feature vector of each method.

After obtaining all the similarity features, the joint feature vector is generated by combining all features together for training (clone type known source codes) and testing (unknown clone type source code) datasets.

3.8 Generating joint feature vector

Only CC types of a matching tuple in the derivation tree are considered in this technique. Developers will occasionally embed a CC type within a larger context. The parents may not be recognised as clones due to considerable differences in the surrounding nodes. The four step of characteristic vector creation, known as vector merging, is used to detect CC types by summing the indices of specific vertex patterns

$$E = \langle A\&B\&C\&D \rangle \quad (7)$$

E' = Joint feature vector; A = lexical similarity; B = Semantic Similairity; C =Structural similarity; D = Syntactic similarity;

Using the aforementioned Eq. (2), a sliding frame is moved along a fictionalised form of the parse tree, and the windows are adjusted so that the combined column has a sufficient CC. The decision of which node in the tree to merge is an important one since these nodes will take better boundaries among cloned CCs rather than containing big sub trees. For merging vectors, the origins of statement trees, which are equivalent to quantum units for duplicate, will be a superior alternative. These selected nodes are known as joint feature vectors. These joint feature vectors are used to build the training data for detecting CCs from known source code.

3.9 CCD using joint feature vector

From the joint features, the similarity features are collectively joined together. The LV-mapper, which is derived from the source code, recognises all of these properties. The source code is compiled into training and testing data. Now these training and testing data are enhanced and converted together to calculate the distance between the both for better identification of clone and its types.

The Euclidian distance is used to detect the clone type by calculating the distance between the training (known source code) and the testing (unknown source code). In which the training data is derived from the joint feature vector and testing data is taken from the given source code. The clone type is calculated for four similarity features which are directly compared as the n-dimensional vectors. The Euclidean distance for given similarity features are calculated as

$$d(p, q_i) = \min\left(\sqrt{\sum_{i=1}^{n} q_{i\,(1)} - p_i},\ \sqrt{\sum_{i=1}^{n} q_{i\,(2)} - p_i}, \ldots\ldots\ldots, \sqrt{\sum_{i=1}^{n} q_{i\,(m)} - p_i}\right) \quad (8)$$

From Eq. (8), $q_{i_{(m)}}$ is i^{th} join feature vector of m^{th} type clone from training data and p_i is the i^{th} join feature vector of test data. The distance value below threshold is identifies as clone. The unique feature vector extracted for each clone type.

Algorithm for the detection of CC types

Step 1: The training and testing data is splitted up from the original source code data.

Step 2: The CBs are extracted using TXL and it is tokenized using flex tool.

Step 3: The hash indexing has been used to detect the CPs by collecting all seeds (tokens), converting them to hash value and indexing them in a hash table.

Step 4: From the detected clone pairs, the lexical features (Type -I) is extracted by using the mechanism of CBOW tool (a type of word2vector).

Step 5: The synonyms are extracted from the CPs using word-net tool (sysnet) and extracted synonyms are urged to detect the semantic features (Type -II) with the help of word2vdector (shallow neural network).

Step 6: The structural CC (Type -III) is extracted directly from the source code by using software metrics like CZM and OOM.

Step 7: Again from the source code program, the syntactic features (Type IV) has been extracted using AST model.

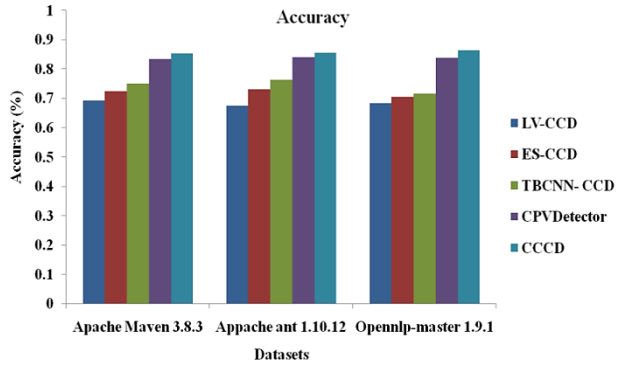

Figure. 3 Evaluation of accuracy

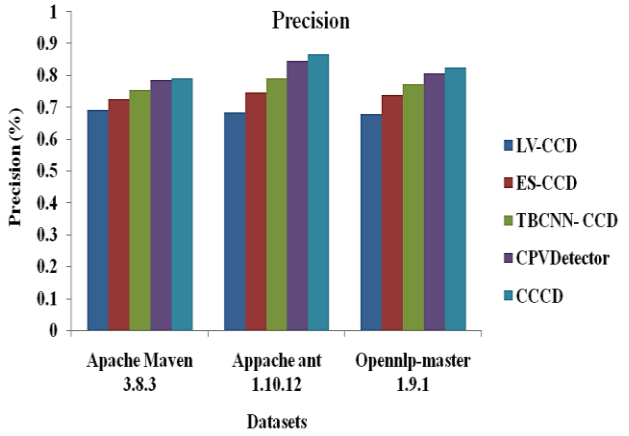

Figure. 4 Evaluation of precision

Step 8: These determined similarity features are integrated together in a joint feature vector for the identification of CC types.

Step 9: By using the Euclidean distance Eq (6), the clone type are identified.

4. Performance evaluation

The clone type known dataset is used for generating training feature vector. A benchmark dataset BigCloneBench is used to generate the training dataset. This is consisting of a large number of manually approved clones from the IJaDataset-2.0 source.

The testing datasets details are:

Apache maven 3.8.3 is a toolkit for managing and comprehending software projects. Maven depends upon the presumption of a Project Entity Paradigm (POM), can manage a project's development, monitoring, and information from a primary source of contact [23].

Appache ant 1.10.12 is a Java library and function tool that manages operations indicated in construct documents as objectives and extensibility endpoints. The well-known use of Ant is the creation of Java applications, as it has a collection of built-in tasks for compiling, assembling, testing, and executing Java programs [24].

Appache opennlp-master 1.9.1 is a natural language information retrieval toolkit based on machine learning. Among the most common NLP operations supported are indexing, sentence classification, part-of-speech labelling, named entity identification, stacking, and processing [25]

Using these datasets, the proposed method's efficiency is compared to that of existing methods such as LV-CCD [9], ES-CCD [13], TBCNN-CCD [14], and CPVDetector [15] in terms of accuracy, precision, recall, time period, memory space, and clone types for detecting all four types of CC detection and its similarity features.

4.1 Accuracy

It is calculated by dividing the number of properly recognised CCs by the total CCs and non-CCs in the clone repository (actual).

Fig. 3 displays the results of accuracy achieved for proposed and existing methods to detect clone types. For the dataset apache maven 3.8.3, Appache ant 1.10.12 and Opennlp-master 1.9.1, it is observed that the accuracy of proposed model increased 23.37%, 26.62%, and 26.46% than LV-CCD , 17.93%, 17.09% and 22.52% than ES-CCD, 12.35%, 13.22% and 14.41% than TBCNN-CCD, 2.39%, 1.66% and 3.22% than CPVDetector respectively. From this analysis, it is proved that the proposed model attains a higher accuracy than the other existing methods.

4.2 Precision

It is computed by calculating the total number of identified CCs by the No. of accurately detected CCs (predicted).

Fig. 4 displays the results of precision achieved for proposed and existing methods to detect clone types. For the given dataset Apache Maven 3.8.3, Appache ant 1.10.12 and Opennlp-master 1.9.1 , it is observed that the precision of proposed model increased 14.32%, 26.79%, and 21.35% than LV-CCD, 8.66%, 16.09% and 11.65% than ES-CCD, 4.77%, 9.48% and 6.73% than TBCNN-CCD, 1.66%, 2.24% and 2.10 % than CPVDetector respectively. From the analysis, it is proved that the proposed model attains a higher precision than the other existing methods.

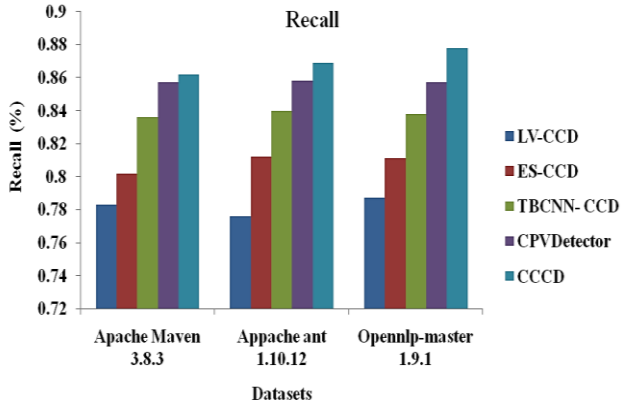

Figure. 5 Evaluation of recall

Table 4. Comparison of time (processing) period and memory storage

	LV-CCD		
k	**2**	**3**	**4**
Time	126 mts	68 mts	38 mts
Memory	1894 mb	979 mb	964 mb
	ES-CCD		
k	**2**	**3**	**4**
Time	114 mts	57 mts	30 mts
Memory	1803mb	966 mb	951 mb
	TBCNN-CCD		
k	**2**	**3**	**4**
Time	103 mts	46 mts	24 mts
Memory	1714 mb	954 mb	938mb
	CPVDetector		
k	**2**	**3**	**4**
Time	94 mts	35 mts	18 mts
Memory	1624 mb	954 mb	925 mb
	CCCD		
k	**2**	**3**	**4**
Time	86 mts	27 mts	12 mts
Memory	1532 mb	942 mb	912 mb

4.3 Recall

The recall is obtained by splitting the total count of CCs in the database by the number of successfully recognized CCs (actual).

Fig. 5 displays the results of recall achieved for proposed and existing methods to detect clone types. For the given dataset Apache Maven 3.8.3, Appache ant 1.10.12 and Opennlp-master 1.9.1, it is observed that the recall of proposed model increased 26.95%, 11.98%, and 13.14% than LV-CCD, 7.48%, 7.02% and 8.26% than ES-CCD, 3.11%, 3.45% and 3.45% than TBCNN-CCD, 0.58%, 1.28% and 2.45% than CPVDetector respectively.

According to the results of the analysis, the suggested model has a higher recall than the existing technique.

Table 5. Comparison of LV-CC types

	LV-CCD		
	Type- 1 & 2	**Type 3**	**ALL**
Apache Maven 3.8.3	259	267	717
Appache ant 1.10.12	2334	471	2591
Opennlp-master 1.9.1	306	474	540
	ES-CCD		
	Type- 1 & 2	**Type 3**	**ALL**
Apache Maven 3.8.3	252	460	708
Appache ant 1.10.12	2325	462	2643
Opennlp-master 1.9.1	294	465	610
	TBCNN-CCD		
	Type- 1 & 2	**Type 3**	**ALL**
Apache Maven 3.8.3	247	454	699
Appache ant 1.10.12	2318	451	2697
Opennlp-master 1.9.1	281	452	678
	CPVDetector		
	Type- 1 & 2	**Type 3**	**ALL**
Apache Maven 3.8.3	242	447	689
Appache ant 1.10.12	2309	439	2748
Opennlp-master 1.9.1	269	437	706
	CCCD		
	Type- 1 & 2	**Type 3**	**ALL**
Apache Maven 3.8.3	238	440	678
Appache ant 1.10.12	2248	427	2675
Opennlp-master 1.9.1	257	420	776

4.4 Processing period and storage AND comparison of LV-CC types

Processing duration and memory storage with various parameterizations, where k = seed length is depicted in Table 4 and 5 pfor the comparison of existing and proposed method for detecting all clone types. Time is represented in minutes (mts) and memory in megabytes (mb) .

5. Conclusion

In this research work, CCCD systems are proposed to detect all types of clones (Type 1 to Type 4) effectively by utilizing the lexical, syntactic, semantic and structural features. The joint feature vector is constructed by combining lexical, syntactic, semantic feature and structural features are termed as training data (known clone type). The testing data (unknown clone type) is taken from the source code where both the data are measured using Euclidean distances to calculate the clone type and its similarity features with less computational complexity. The results proved that proposed model increased average accuracy of 25% than LV-CCD, 18 % than ES-CCD and 14.41% than TBCNN-CCD methods. Machine learning and deep learning will be utilized for CCCD systems instead of Euclidian distance. The supervised learning methods will give precise results.

Conflict of interest

The authors declare no conflict of interest.

Author contributions

Conceptualization, methodology, software, validation, Karthik; formal analysis, investigation, Rajdeepa; resources,data curation, writing—original draft preparation, Karthik; writing—review and editing, Karthik; visualization,; supervision, Rajdeepa.

References

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research", *Queen's School of Computing Tech*, Vol. 541, No. 115, pp. 64-68, 2007.

[2] R. Koschke, I. D. Baxter, M. Conradt, and J. R. Cordy, "Software Clone Management Towards Industrial Application", *Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik*, Vol. 2, No. 2, pp. 21-57 2012.

[3] C. K. Roy, J. R. Cordy and R. Koschke, "Comparison and evaluation of CC detection techniques and tools: A qualitative approach", *Science of Computer Programming*, Vol. 74, No. 7, pp. 470-495, 2009.

[4] N. Saini and S. Singh, "Code clones: Detection and management", *Procedia Computer Science*, Vol. 132, pp. 718-727, 2018.

[5] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software", In: *Proc. of 2008 15th Working Conference on Reverse Engineering*, pp. 81-90, 2008.

[6] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization", In: *Proc. of 16th IEEE International Conference on Program Comprehension*, pp. 172-181, 2008.

[7] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection", *IEEE Access*, Vol. 7, pp. 86121-86144, 2019.

[8] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "CCAligner: a token based large-gap clone detector", In: *Proc. of the 40th International Conference on Software Engineering*, pp. 1066-1077, 2018.

[9] 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.

[10] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering", *Journal of Systems and Software*, Vol. 137, pp. 130-142, 2018.

[11] G. Li, H. Liu, Y. Jiang, and J. Jin, "Test-Based Clone Detection: an Initial Try on Semantically Equivalent Methods", *IEEE Access*, Vol. 6, pp. 77643-77655, 2018.

[12] C. Ragkhitwetsagul and J. Krinke, "Siamese: scalable and incremental code clone search via multiple code representations", *Empirical Software Engineering*, Vol. 24, No. 4, pp. 2236-2284, 2019.

[13] A. Sheneamer, S. Roy and J. Kalita, "An Effective Semantic Code Clone Detection Framework Using Pairwise Feature Fusion", *IEEE Access*, Vol. 9, pp. 84828-84844, 2021.

[14] Y. B. Jo, J. Lee and C. J. Yoo, "Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network", *Applied Sciences*, Vol. 11, No.14, p. 6613, 2021.

[15] J. Guo, H. Li, Z. Wang, L. Zhang and C. Wang, "A Novel Vulnerable Code Clone Detector Based on Context Enhancement and Patch Validation", *Wireless Communications and Mobile Computing*, Vol. 2022, 2022.

[16] J. R. Cordy, "The TXL source transformation language", *Science of Computer Programming*, Vol. 61, No. 3, pp. 190-210, 2006.

[17] Paxson, "Flex_Fast Lexical Analyzer Generator. Berkeley", *CA, USA: Lawrence Berkeley National Laboratory*, 1995.

[18] H. Mittal and D. Mandalika, "WordNet Tool in Natural Language Processing", *CSI Communications*, Vol. 40, No. 7, 2016.

[19] M. Kapdan, "Structural code clone detection methodology using software metrics", *International Journal of Software Engineering and Knowledge Engineering*, Vol. 26, No. 2, pp. 307-332, 2016.
[20] Y. B. Jo, J. Lee, and C. J. Yoo, "Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network", *Applied Sciences*, Vol. 11, No. 14, p. 6613, 2021.
[21] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code", In: *Proc. of the ACM on Programming Languages*, Vol. 3, No. POPL, pp. 1–29, 2019.
[22] D. Chandrasekaran and V. Mago, "Evolution of semantic similarity − a survey", *ACM Computing Surveys* (*CSUR*), Vol. 54, No. 2, pp. 1-37, 2021.
[23] [https://github.com/apache/maven]
[24] [https://github.com/apache/ant].
[25] [https://github.com/apache/opennlp].