Code Clone Detection in C Language Programs using Supervised Learning

Author

Qurat Ul Ain

FALL 2017-MS-17(CSE) 00000205526

MS-17 (CSE)

Supervisor

Dr. Wasi Haider Butt

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

ISLAMABAD

31,12, 2019

Code Clone Detection in C Language Programs using Supervised Learning

Author

Qurat Ul Ain

FALL 2017-MS-17(CSE) 00000205526

A thesis submitted in partial fulfillment of the requirements for the degree of

MS Software Engineering

Thesis Supervisor:

Dr. Wasi Haider Butt

Thesis Supervisor's Signature: _____

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY, ISLAMABAD

31,12, 2019

DECLARATION

I certify that this research work titled *"Code Clone Detection in C Language Programs using Supervised Learning"* is my own work under the supervision of Dr. Wasi Haider Butt. This work has not been presented elsewhere for assessment. The material that has been used from other sources has been properly acknowledged / referred.

_____

Signature of Student

Qurat Ul Ain

FALL 2017-MS-17(CSE) 00000205526

LANGUAGE CORRECTNESS CERTIFICATE

This thesis is free of typing, syntax, semantic, grammatical and spelling mistakes. Thesis is also according to the format given by the University for MS thesis work.

_____

Signature of Student

Qurat Ul Ain

FALL 2017-MS-17(CSE) 00000205526

_____

Signature of Supervisor

COPYRIGHT STATEMENT

ACKNOWLEDGEMENTS

I thank Almighty Allah (swt) my Creator Allah for his ultimate guidance throughout my research. Nothing would have been possible without his profound blessing. For all praise is due to God, the Sustainer of all the worlds. Also my admirations be upon Prophet Muhammad (PBUH) and his Holy Household for being source of guidance for people.

I am profusely thankful to my beloved parents who raised me when I was not capable of walking and continued to support me throughout in every department of my life.

I would like to show gratitude to my supervisor **Dr. Wasi Haider Butt** for his tremendous support and cooperation whose constant motivation, persistent efforts and uninvolved words of wisdom ever proved a lighthouse for me. Despite his never ending commitments, he did never mind giving his maximum whenever I requested for his time and support.

I would also like to thank my Guidance Committee Members **Dr. Arsalan Shaukat** and **Dr. Urooj Fatima** for being on my thesis guidance and evaluation committee. Their recommendations are very valued for improvement of the work. I appreciate their guidance throughout the whole thesis.

Finally, I would like to express my gratitude to all the individuals who have rendered valuable assistance to my study.

Dedicated to my exceptional parents, family and friends whose tremendous support and cooperation led me to this wonderful accomplishment

ABSTRACT

Code cloning refers to the duplication of source code. It occurs as a result of copy paste activity without or with minor modification into another section of code. It is the most common way of reusing source code in software development. Several studies suggested that almost 20-50 percent of large software systems consist of cloned code. If a bug is identified in one segment of code, all the segments similar to this need to be checked for the same bug. Consequently, this cloning process may lead to bug propagation that significantly affect maintenance cost. By considering this problem, Code Clone Detection (CCD) appears as an active area of research. Several tools and techniques are introduced so far, for the detection of code clones from various programming languages. However, most of them are unable for the detection of most difficult type of clones semantic or Type 4 clones. Few tools or techniques that can detect these clones utilize traditional methods which can detect type 4 clones with low accuracy. From literature we find few (3 or 4) studies that tried their best to detect all types of clones including type 4 clones with good results (accuracy, execution) but their capabilities are limited to java code because the compilers or parsers utilized by these approaches work for java code only. However, current approaches are inadequate to detect semantic clones along with other (type 1, type 2 and type 3) three types of clones with good results in programing languages (e.g. C/C++).

In this research work we attempt to improve the accuracy of semantic or type 4 clones while not compromising the accuracies of other three types of clones in C programs. For this purpose, we conduct an experiment by utilizing 2 datasets (Krawitz and Roy et al.). Different from manually defining features for code clone detection, our framework can automatically extract features by analyzing abstract syntax trees (ASTs) of source code. Afterwards, supervised learning based classification model is used and conduct 2 sets of experiment for code clone detection. Each set consists of pair instance feature using linear combination. The classification model is trained and tested using different types of validations. Furthermore, to check the effectiveness of proposed framework if a non-clone occurs in the dataset, we manually add some non-clones and iterate the whole process.

The performance of our framework is compared with state of the art and popular code clone detection approaches that are used in several recent studies. Results indicate that the proposed framework is superior in the detection of Type 4 clones and comparable in finding Type1 clones.

However, our framework does not give acceptable results in finding Type2 and Type3 clones. Therefore, we perform some extended experiments and get valuable results on all types of clones.

KEYWORDS

Code clones, Code Clone Detection, C Source Code, Abstract Syntax Tree, Feature Extraction, Random Forest,

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

CHAPTER 1: INTRODUCTION

In this chapter the comprehensive information of the research work is provided which is divided into the following sub sections. **Section 1.1** gives the overview of code clone detection. **Section 1.2** consists of the problem statement being addressed in this work. Research flow that is followed to accomplish this research work is explained in **Section 1.3**. **Section 1.4** explain the research contribution and thesis organization is given in **Section 1.5**.

1.1. Overview

Code duplication by copying and pasting without or minor modification into another section of code frequently occurs in software development. This copied code is called code clone and the process is called code cloning. Various studies suggested that almost 20-50 percent of large software systems consist of cloned code [1] [2]. If an error is identified in one part of the code, correction is required in all the replicated segments. Therefore, it is essential to identify all related segments throughout the source code.

1.1.1. Background

1.1.1.1. Code Clone Detection

There is no appropriate definition of code clone. Different researchers used different terms for cloning. Krinke [3] utilized the term “similar code”. Baxter et al. [4] suggested that a clone is “a code segment that is identical to another segment”. Ducasse et al. [5] utilized the term “duplicated code”. Komondoor and Horwitz [6] also used “duplicated code” and clone as an item of duplicated code.

Basic types of clones are listed below [7]:

Exact clones (Type 1)

Identical code segments except for changes in comments, layouts and whitespaces are known as exact clones or type 1 clones.

Renamed clones (Type 2)

Code segments which are syntactically or structurally similar other than changes in comments, identifiers, types, literals, and layouts. These clones are also called parameterized clones.

Near Miss clones (Type 3)

Copied pieces with further modification such as addition or removal of statements and changes in whitespaces, identifiers, layouts, comments, and types but outcomes are similar. These clones are also known as gapped clones.

Semantic clones (Type4)

More than one code segments that are functionally similar but implemented by different syntactic variants are called semantic or type 4 clones.

Although there are four types of clones, sometimes people use different terms when referring to the clone relation to their experiments. Common terms utilized by them are given below.

Structural Clones

Simple clones that follow the syntactic structure of a particular language within the syntactic boundary. These boundaries can be statement boundary, structure boundary, class boundary etc. A structural clone can be any of the four types of clones depending on its similarity level.

Function Clones

These clones are simple clones that are limited to the procedure or method/function level granularity. Similar to the structural clones these clones can also be any of the four types of clones based on their level of similarity.

Cloning is beneficial but it can also be harmful in many ways. For example, in many software engineering tasks such as aspect mining, program understanding, plagiarism detection, copyright infringement investigation, code compaction, software evolution analysis, code quality analysis, bug detection and virus detection may need the extraction of semantically or syntactically similar code blocks, making clone detection effective and useful part of software analysis [7]. They can also lead to the bug propagation that significantly increases the software maintenance cost. By considering these maintenance problems, software clone detection appeared as an active area of research. Several approaches and tools introduced so far, for the detection of code clones and there have been many comparisons and evaluations studies. Text-based approaches, Token-based approaches, Tree-based approach, Metric based, Semantic approaches and Hybrid approaches are mainly used [8] [9]. Tools include NICAD [10] [11], CCFinderX [12] [13], Simian [14], CPMiner [15] etc. Furthermore, certain similarity measure algorithms such as Fingerprinting [16] [17],

Neural Networks [18], Euclidian Distance [19] etc. are also utilized for the detection of code clones.

1.1.1.2. Supervised Learning

Supervise learning classification model utilized in proposed approach is random forest.

Random Forest

Random forest is a supervised learning classification model, for the classification of dataset it utilizes decision trees. Growing an ensemble of trees and deciding the type of class by voting, significantly improves the classification accuracy. For growing these ensembles random vectors are constructed. Each tree is generated from one random vector. Random forest consists of classification trees. By analyzing output of these trees the classification problems are solved. The random forest prediction is determined by majority voting [20].

1.2. PROBLEM STATEMENT

Several techniques and tools are developed to detect code clones from various programing languages, but most of them are unable for the detection of most difficult type of clones semantic or Type 4 clones. Few tools or techniques that can detect these clones utilize traditional methods which can detect type 4 clones with low accuracy [9]. From literature we find few (3 or 4) studies that tried their best to detect all types of clones including type 4 clones with good results (accuracy, execution) [48][67][71][82] but they are applicable to java code only because parsers or compilers used in these studies are limited to java. However, current approaches are inadequate to find semantic clones along with other (type 1, type 2 and type 3) three types of clones with good results in programing languages (e.g. C/C++).

1.3. RESEARCH FLOW

The research process is performed in a systematic way as shown in **Figure 1.1**. The first step of any research is the identification of the problem. After the identification of the problem, we proceed to the next stage which is problem solving. For the solution of the problem a comprehensive literature review is performed in a systematic way. This literature review also covers the work related to the proposed solution.

Moreover, the proposed solution presents an approach for the detection of all types of (mainly type 4) clones in C source code. To measure the similarity between two code segments, we parse these segments into their constitutional segments in the form of Abstract Syntax Trees(ASTs). In the next step, features are extracted from ASTs and combine them linearly to generate a dataset for training and testing of classification model. The classification model utilized in this approach is random forest. After the implementation, results are validated and comparison is performed with other code clone detectors. Furthermore, the whole research work is discussed and its limitations are analyzed. The final step concludes the research work and suggests the future work.

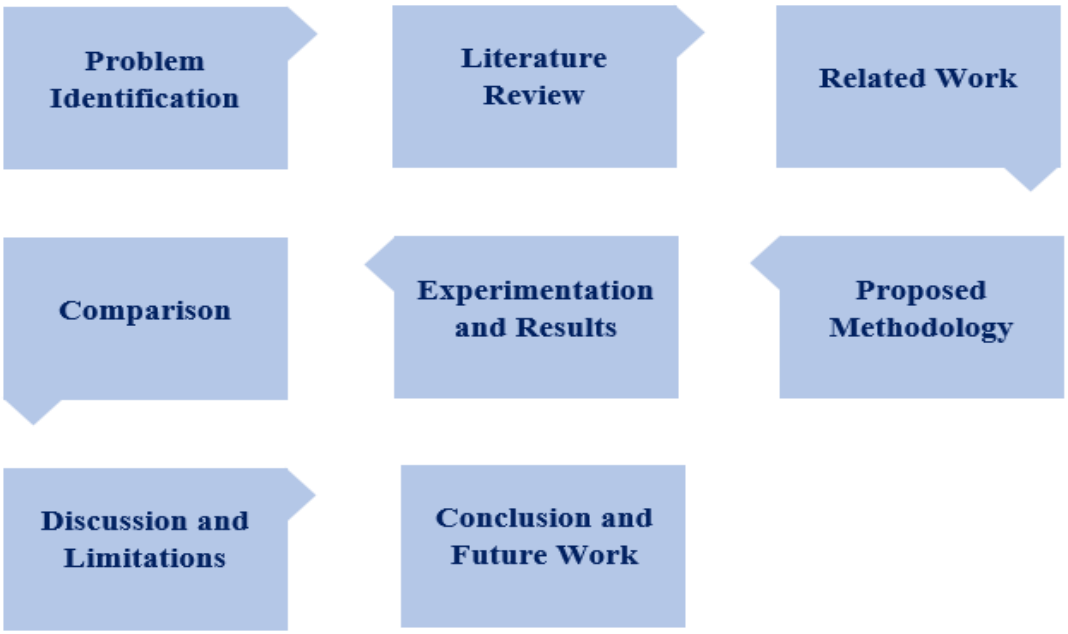


Figure 1.1: Research Flow

1.4. RESEARCH CONTRIBUTION AND MAIN OBJECTIVES.

This work is carried out to develop an approach that can improve the results of semantic or type 4 clones including other three types from C code. Supervised machine learning is used for this purpose. Contribution of this research and its main objectives are given below.

- We utilized features of ASTs for identification of syntactic (type1, type2 type 3) and semantic (type 4) clones. For this purpose, we extract features from both normalized ASTs and Original ASTs to get better accuracy in C code. To the best of our knowledge we are first to utilize such kind of features from C source code.

- We present a pair of code fragments as a vector in a linear way to enhance the detection of code clones in C.
- We utilize these features to learn a classification model to detect code clones by using random forest, a supervised machine learning classification model.
- Our approach is compared with other code clone detectors and prove that our approach overall gives higher accuracy.

1.5. THESIS ORGANIZATION

Organization of this research work is explained with the help of **Figure 1. *2***. **Chapter 1** demonstrates the detailed introduction of code clone detection which consist of overview, problem statement, research flow, research contribution and main objectives, and thesis organization. **Chapter 2** presents the comprehensive literature review which focus on the work done in the field of code clone detection by various scholars and researchers. It consists of three sections. First is review protocol which explains how the research is conducted, in second section entire research done in the field of code clone detection is discussed while third section refers to the research gaps. **Chapters 3** presents the proposed methodology for the identified problem statement, where each step of this methodology discussed in detail. **Chapter 4** covers experimentations and results. Detailed about datasets and whole classification process are discussed in this section. Comparison with other approaches or detectors also discussed in this section. **Chapter 5** discuss the entire thesis along with limitations. **Chapter 6** finally concludes the research work and future work is suggested.

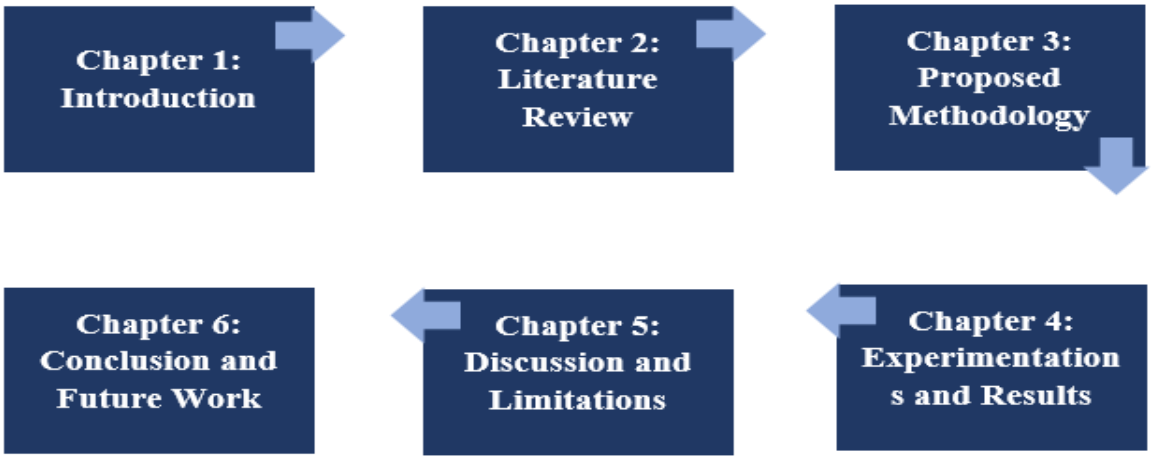


Figure 1. 2: Thesis Organization

Chapter 2

Literature Review

CHAPTER 2: LITERATURE REVIEW

In this chapter detail literature review on code clone detection is conducted. In **Section 2.1** review protocol is presented. **Section 2.2** consists of the results we get from review protocol and, gaps which form a base of our research are discussed in **Section 2.3**.

2.1. REVIEW PROTOCOL

Review protocol consists of five elements. These five elements include category definition, selection and rejection criteria, search process, quality assessment, data extraction and synthesis. The detail of the remaining five elements provided in subsequent sections.

2.1.1. Categories Definition

For the simplification of data extraction and synthesis process, we define six categories. The description of these categories is given below.

Textual Approaches: Code clones can be detected by using different CCD approaches. One of them is textual approaches, they detect type1 clones more effectively [68]. However, these approaches can also identify type2 and type3 clones [8]. Therefore, this category consists of research studies that particularly deals with CCD using textual approaches.

Lexical Approaches: The research studies that particularly deal with CCD based on lexical approaches are placed under this category. Lexical approaches are also known as token-based approaches and able to identify type 2 clones efficiently [68]. However, they can also uncover type1 and type3 clones [8].

Tree-Based Approaches: This category consists of the studies that are dealing with CCD using tree-based approaches. They are most effective for the detection of type3 clones [68]. However, they have the ability to detect type1, type2, and type4 clones [8].

Metric Based Approaches: The research studies in which code clones are detected by utilizing metric-based approaches are placed under this category. They can detect type3 clones effectively [68]. However, they can also uncover type1, type2 and type4 clones.

Semantic Approaches: The research studies that particularly deal with CCD based on semantic approaches are placed under this category. A semantic approach similar to tree-based and metric-based approaches have the ability to detect type1, type2, type3 and type4 clones. They are mainly used to uncover semantic or type 4 clones [68].

Hybrid Approaches: The research studies in which a combination of two or more aforementioned techniques e.g. (Textual, Lexical, AST based, Metric Based or Semantic) is utilized should be placed under this category.

2.1.2. Selection and rejection criteria

We define selection and rejection criteria to carry out this literature review for obtaining desired goals. For this purpose, some rules are defined as discussed below.

- The research studies in which keyword "code clone detection" is included in the title or abstract are selected. We discard such research studies where code clone detection (CCD) is partially or not discussed.
- We consider the conference papers that are published from 2015 to 2019 and journals published from 2013 to 2019. All those research studies published before 2013 are rejected to assure the inclusion of the latest research studies.
- To perform this literature review, four well-known scientific databases (i.e. Springer, IEEE, Elsevier and ACM) are selected. Therefore, the research studies that are published in one of the above-mentioned databases are considered. Studies other than these repositories are not selected.
- Selected studies must be result oriented. Some solid evidence and experimentation must support the proposed methodologies and their ultimate outcomes.
- The research papers that have almost similar contents are discarded and only one of them is selected.

2.1.3. Search Process

The search process is started by utilizing four databases (IEEE, ACM, Springer and Elsevier) as described in selection and rejection rules. We have utilized many search terms or keywords while performing the search process. The overall summary of the search process is given in **Table 2.*1***. To carry out the research process two types of operators such as AND, OR are utilized. The outcomes collected from AND operator are not enough that is why OR operator is used. However, the results obtained by using the OR operator are very large, it is not feasible to scan all of these results. Therefore, advanced search options are utilized, provided by selected databases e.g. where keyword contain "time span" in order to get precise results.

Table 2.1: Summary of search terms with results.

Sr.#	Search terms	Operator	IEEE	ACM	Elsevier	Springer
1	CCD	N/A	201	1385	25375	20134
2	CCD, Text based Techniques	AND	5	88	161	113
		OR	2933	5274	15780	20245
3	CCD, Token based techniques	AND	4	864	174	103
		OR	378	5155	3610	2377
4	CCD, Tree based techniques	AND	8	877	2327	1748
		OR	3542	5228	13214	18331
5	CCD, Metric based techniques	AND	14	878	1018	665
		OR	7879	5245	3571	8987
6	CCD, PDG based techniques	AND	4	63	39	23
		OR	204	5154	9024	8759
7	CCD, Hybrid techniques	AND	8	38	19	79
		OR	1745	1847	1789	3358
8	CCD Tools	AND	111	325	413	389
		OR	5438	2646	9734	6559
9	CCD, Machine learning techniques	AND	10	539	818	564
		OR	4400	3734	6132	6377

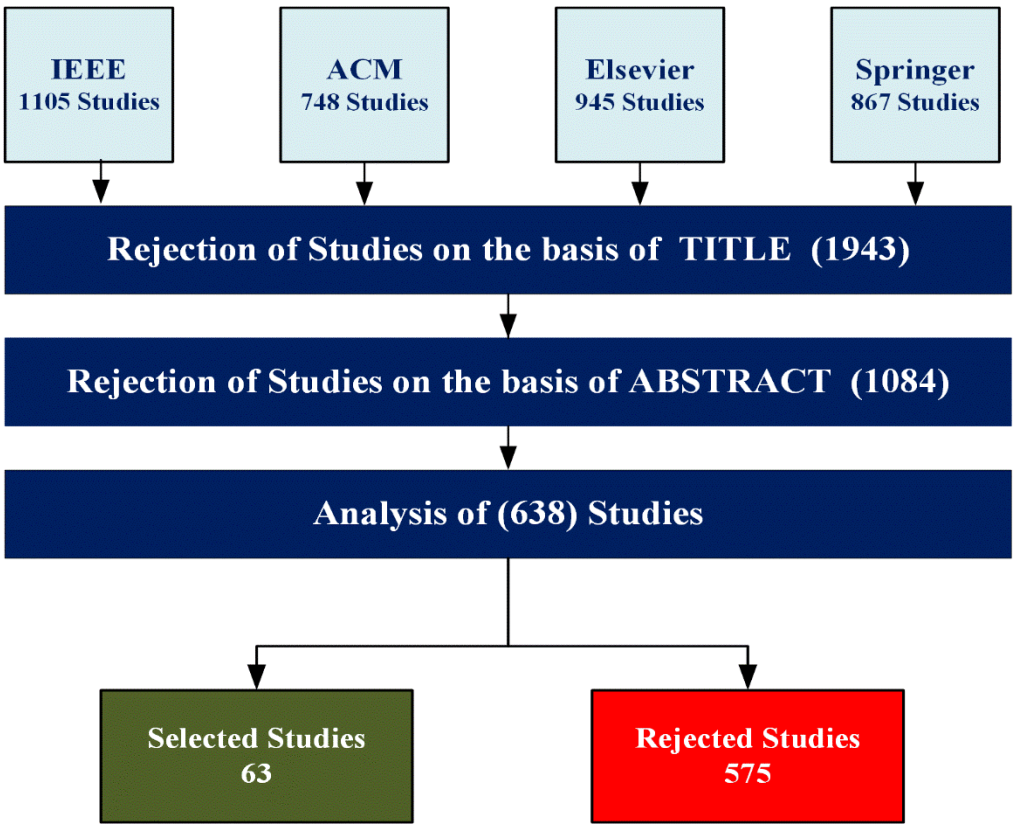


Figure 2.1: Summary of the Search Process.

After the investigation of primary results, we select only those studies that are highly related to CCD. Finally, we get 63 research studies by following certain steps **Figure 2.1**.

- We overall consider 3665 research papers and reject 1943 by reading their title.
- Afterword's we consider remaining 1722 papers and reject 1084 by examining their abstract.
- Then we investigate the remaining 638 research studies. Based on this investigation we exclude 575 research studies and include 63 studies, which are totally according to our defined criteria.

2.1.4. Quality Assessment

To assure the reliable results of this literature review, we have selected high impact studies e.g. researches from repositories that are authentic and accepted all over the world. Forty-five (45) research studies are selected from IEEE, eight (8) studies from Elsevier, five (5) studies from ACM and four (4) studies from Springer. The results presented in **Table 2.2** indicate that we try our best to choose the high impact and the latest research studies. The overall summary of the repositories w.r.t their publication type is given in **Table 2.2**. **Database** represents the names of the repositories. **Type** represents that whether the selected research study belongs to either journal or conference. **References** are given for selected studies. **Total** represents the number of total conference or journal papers of every scientific repository.

In **Table 2.2**, it can observe that 40 conference papers and 5 journal papers selected from IEEE, 5 conference papers selected from ACM, 2 conference papers and 6 journal papers from Elsevier and 2 conference papers and 2 journal papers selected from Springer. We select papers from 2013 onward. We consider all journal papers published from 2013 to 2019.

There is no journal paper related to CCD available in 2013, 2 papers found published in 2014, 2 papers published in 2015, 1 study found published in 2016, 1 studies in 2017, 3 studies published in 2018 and 4 studies published in 2019. The journal papers represented by a brown bar in **Figure 2.2**. Similarly, conference papers published from 2015 to 2019 are selected. 5 conference papers found published in 2015, 11 papers published in 2016, 20 studies found published in 2017, 11 studies published in 2018 and 2 papers are published in 2019 as represented by a blue bar in **Figure 2.2**.

Table 2.2: Summary of selected studies according to scientific databases and publication type.

Database	Type	Reference	Total
IEEE	Conference	[21][23][24][25][26][27][30][31][32][33][34][39][40] [41][42][43][44][45][47] [49][51][53][54] [58][59][62] [63][64][65][66][67][68][69][70] [74][75][77][78][80][82]	45
	Journal	[46][48][56][60][81]	
ACM	Conference	[28][38][50][52][72]	5
	Journal	Nil	
Elsevier	Conference	[55][76]	8
	Journal	[22][35][36][37][57][71]	
Springer	Conference	[61][79]	4
	Journal	[29][73]	

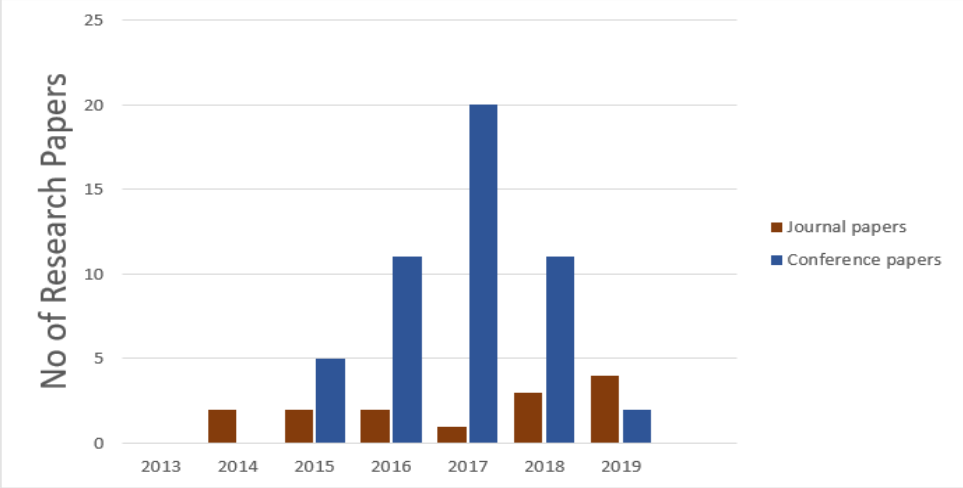


Figure 2.2: No. of selected studies w.r.t publication year

2.1.5. Data Extraction and Synthesis

We have developed a template to extract data and perform synthesis as presented in **Table 2.3**. Firstly, the extrication of bibliographic information of each selected study is performed. After that, core findings such as the proposed methodologies and implementation details of each selected study are extracted. In order to achieve the goals of literature, this provides the basis to carry out a detailed analysis.

Table 2.3: Data extraction and synthesis template

Sr.#	Description	Details
1	Bibliographic Information	Title, publication year and type of research paper (i.e. Conference or Journal) is observed.
2	Proposed methodology	Methodology followed by each study is analyzed.
3	Implementation details	Technologies used to implement the proposed methodology are analyzed.
4	Outcomes	Outcomes of each selected study are thoroughly analyzed.
5	Grouping	Selected categories are arranged in groups. The results are summarized in **Table 2.4.**
6	Investigation of categories	Analysis of each category to find the answers of the RQ's. The results are summarized below: Textual Approaches **Table 2.5**, Lexical Approaches **Table 2.6**,Tree Based Approaches **Table 2.7**, Metric Based Approaches **Table 2.8**, Semantic Approaches **Table 2.9**, Hybrid Approaches **Table 2.10.**
7	Open Source Subject Systems	Source code of various Open Source Subject utilized in selected studies are examined in **Table 2.11**.

Table 2.4 : Classification results of selected studies.

Sr.#	Category	References of Corresponding Studies	Total
1	Textual	[21][22][23][24][25][26][27][28][29][30][31][32][33][34]	14
2	Lexical	[35][36][37][38][39][40][41][42][43][44]	10
3	Tree Based	[45][46][47][48]	4
4	Metric Based	[49][50][51][52][53][54][55][56][57]	9
5	Semantic	[58][59][60][61][62][63][64]	7
6	Hybrid	[65][66][67][68][69][70][71][72][73][74][75][76][77][78] [79][80][81][82]	18

2.2. RESULTS AND ANALYSIS

The main objective of this literature is to examine the given literature according to the research questions. Out of 62 research studies, 13 are published as journals and 49 are published in international conferences. The focus of these studies, published as journals or conferences on

CCD. Studies related to CCD are published in wide-range of conference and journal proceedings. It can be noticed that journals like Journal of Network and computer application, Expert system with applications, IEEE transaction on software engineering, IEEE access, Computer and Electrical engineering, the journal of system and software and Journal of computer science and technology are highly contributing to our research. Similarly, there is a wide variety of conferences such as conferences like International conference on software engineering, a conference on Software Analysis, Evolution, and Reengineering contribute largely to our studies. Almost 79% of our literature published in conferences and 21% published in journals. These research studies divided into six categories as presented in **Table 2.4**. For further analysis, references of corresponding studies given against each category.

It can be seen that in **Table 2.4** the Textual Approaches consist of fourteen studies, Lexical Approaches comprises ten research studies, Tree-Based Approaches consist of four studies, Metric based Approaches comprises nine studies, Semantic Approaches comprises seven research studies, and Hybrid Approaches contain eighteen studies. The detail of these categories summarized in subsequent sections.

2.2.1. Textual Approaches

Several CCD techniques depend on text-based techniques. These techniques consider the source code as a sequence of lines or strings. To find the sequences of the same lines, two code pieces are compared with each other. Whenever at least two code fragments found to be similar then by detection technique they are returned as clone class or clone pair. No or little transformation is done with source code because these are purely text-based techniques. In **Table 2.5**, CCD based on Textual approaches is analyzed with the parameters given below.1) **Language** describes the language of source code that is used for clone detection. 2) **Input Type/Intermediate state** shows that input taken by clone detection technique or intermediate format in which source code transform before clone detection. 3) **Algorithm/Classifier used** indicates that algorithms/Classifiers utilized for the identification of clones. 4) **Clone Type Detected** shows types of clones detected in these studies. The summary of these research studies is given below.

In **Table 2.5,** Ragkhitwetsagul and Krinke [21] utilize compilation/decompilation to enhance clone detection. For this purpose, they use NICAD, a text-based code clone detector, java source code as input and uncover type1, type2 and type3 clones. Kim and Lee [22] introduce Vuddy, a

scalable approach for vulnerable clone discovery by utilizing C/C++ programs as source code and generate fingerprints that are utilized as input. This approach has the ability to identify type1 and type2 clones. Jadon [23] proposes a technique for the detection of similar clones (type 3) and quantify their similarity. The proposed technique detects similar clones (type 3) by using C programs as source code, feature set as an intermediate format and Support Vector Machine (SVM) for classification of the algorithm. Yu et al. [24] propose multigranuality CCD method based on Java bytecode by utilizing Java source code that transforms into the .txt format and uncovers type1, type2 and type3 clones. Kim et al. [25] present Vuddy, a scalable approach for vulnerable code clone discovery. The presented approach utilizes C/C++ programs as source code and generate fingerprints from this source code, which are utilized as input for CCD. MD5 Hash algorithm is used to produce hash values. It can detect type1 and type2 clones.

Nakamura et al. [26] introduce an approach to detect interlanguage clones for a multilingual web application. Authors utilize source code of multiple programming languages. Pattern mining is used to identify frequently co-used programming languages. This approach is able to detect type3 clones. Lyu et al. [27] propose SuiDroid, an approach for android app clone detection. It is implemented by using python and shellcode. SuiDroid utilizes Layout XML files to identify the apps, layout trees as intermediate representation and CTPH Hash algorithm to measure the similarity. Results indicate that type1, type2 and type3 clones are identified. Xue et al. [28] describe a novel framework, clone hunter that integrates machine learning based binary CCD to speed up the elimination of redundant array bound checks in binary executables. They utilize assembly code as source code and Feature vectors as an intermediate format. AP Clustering algorithm is used for binary CCD. This framework uncovers type1, type2 and type3 clones. Chen et al. [29] apply NICAD, a text-based code clone detector for detecting android malware. For this purpose, Java source code is used as input and as we know that NICAD can detect type1, type2 and type3 clones so we can assume that these types of clones are identified. Thalle et al. [30] describe the results from the analyses of code clones in real-world PLC software. These results show that normalized C/C++, ST source code is utilized and type1 and type2 clones are detected. Newman et al. [31] develop a tool, srSlice. It utilizes C/C++ source code, which is transformed into srcML as an intermediate format, and uncover code clones. Liu et al. [32] propose VEDFECT a vulnerable code clone system. For this purpose, C/C++ programs utilize as input, MD5 Hash algorithm that is applied on code blocks, (which are different from preprocessed code blocks) to

construct fingerprints. By matching, the preprocessed code blocks with fingerprints VEDEFECT uncovers the vulnerable code clones.

Reddivari and Khan [33] developed a code clone tool named CloneTM based on LDA. It supports multiple programming languages including java and C++. This tool is evaluated on two systems industrial proprietary system WDS and open source system iTRUST. Ghosh et al. [34] propose an approach for the detection of semantic clones with the help of source code comments. The dataset used for this purpose, consist of java code and code clones are detected by using LDA. Results indicate that using LDA in the presence of comments we get better precision and recall than that of GRAPLE in the presence of PDG.

Table 2.5: Summary of research studies using Textual Approaches.

Sr.#	Reference No	Language	Input Type/ Intermediate State	Algorithm /Classifier Used	Clone Type Detected
1	[21]	Java	Source Code	N/A	1,2,3
2	[22]	C/C++	Fingerprints	N/A	1,2
3	[23]	C	Feature Set	SVM	3
4	[24]	Java	.txt files	N/A	1,2,3
5	[25]	C/C++	Fingerprints	MD5 Hash	1, 2
6	[26]	Multiple e.g. (HTML,Javascript)	Source Code	N/A	3
7	[27]	Layout XML Files	Layout Trees	CTPH Hash	1,2,3
8	[28]	Assembly	Feature Vector	AP Clustering	1,2,3
9	[29]	Java	N/A	N/A	1,2,3
10	[30]	C/C++,ST	Normalized Source code	N/A	1,2
11	[31]	C/C++	srcML	N/A	N/A
12	[32]	C/C++	Preprocessed Source code	MD5 Hash	N/A
13	[33]	Multiple e.g (java and C++)	Source Code	N/A	N/A
14	[34]	Java	Source code comments	LDA	4

2.2.2. Lexical Approaches

Lexical approaches are also known as token-based approaches. These approaches consist of two steps, lexical analysis and clone detection. They transform targeted source code into a sequence of tokens with the help of laxer or parser. The sequence of tokens is scanned to find duplicate subsequences of tokens and finally, the original code fragment that represents the duplicate subsequences will be returned as clones. In **Table 2.6**, CCD based on Lexical approaches is analyzed with the parameters given below. 1) **Dataset** describes the datasets available in these studies, which are converted into tokes. 2) **Data Structure** evaluates the data structure used for clone detection. 5) **Algorithm Used** indicates algorithms utilized to measure similarity. 5) **Clone Type Detected** evaluates types of clones detected in these studies. The summary of these research studies provided in subsequent paragraphs.

In **Table 2.6**, Nishi and Damevski [35] introduce a clone detection approach by applying adaptive prefix filtering heuristic. It utilizes IJaDataset 2.0, a clone detection benchmark. As a data structure Delta inverted index is used for retrieving matching documents. This approach is able to find type1, type2 and type3 clones. Tekchandani et al. [36] present git code clone genealogy extraction model by utilizing the DAG data structure and can detect type1 and type2 clones. Farhadi et al. [37] present scalclone, a scalable assembly code clone search system by using Zlib, DLL18, Malware297 and DLL1GB datasets. LSH algorithm is applied to find inexact clones. It can detect type1, type2 and type3 clones. Wang et al. [38] develop CCAligner, a token based clone detector. It employs C, Java files as a dataset, find type1, type2, and type3 clones. Yuki et al. [39] present a technique to detect multi-grained code clones. In the presented technique, Java files utilized as a dataset and Smith-Waterman algorithm utilized to identify the identical hash sequence. It uncovers type1, type2 and type3 clones. Sajnani et al. [40] propose SourcererCC, a token based clone detection tool. It employs IJaDataset. The inverted index data structure is applied to quickly query the proportional clones of a given code block. To measure recall, two benchmarks are used: 1) BigCloneBench, a benchmark of real clones, 2) Mutation/Injection based, the framework of thousands of artificial fine-grained clones. It can identify type1, type2 and type3 clones. Similarly, Semura et al. [41] develop another clone detection tool CCFinderSW. It takes dataset from Rosetta Code, a webpage that provides source code implemented in various programming languages, and uncovers type1 and type2 clones. Li et al. [42] present CCLEARNER, a deep learning based clone

Table 2.6: Summary of research studies using Lexical Approaches.

Sr.#	Reference No	Dataset	Data Structure	Algorithm Used	Types of clones detected
1	[35]	IJaDataset 2.0	delta inverted index	N/A	1,2,3
2	[36]	N/A	DAG	N/A	1,2
3	[37]	Zlib, DLL18, Malware297, DLL1GB	N/A	LSH	1, 2,3
4	[38]	C and Java files	N/A	N/A	1,2,3
5	[39]	Java files	N/A	Smith-Waterman	1,2,3
6	[40]	IJaDataset	Inverted Index	N/A	1,2,3
7	[41]	from Rosetta Code	N/A	N/A	1,2
8	[42]	IJaDataset	N/A	Deep Neural Network	1,2,3
9	[43]	multiple (c, cobol85, cpp14, ecmasript, java9, python3)	N/A	N/A	1,2,3
10	[44]	Java	N/A	N/A	Method level (1,2,3or 4)

detection approach. This approach utilizes IJaDataset (java code) that transforms into tokens. Deep learning algorithm used by this approach is DNN and uncover type1, type2 and type3 clones. Semura et al. [43] propose a technique that can automatically extract lexical information from grammar definition of the parser generator which is necessary for token based detection of code clones. They also extend CCFinderSW, a clone detection tool that has a lexical information extractor from grammar definition of ANTLER which is a parser generator. This technique can detect type1, type2 and type3 clones from multiple languages (c, cobol85, cpp14, ecmasript, java9, python3). Uemura et al. [44] propose an integrated approach for code clone detection and tracking their histories using historage. To demonstrate the histories of code clones are analyzed in this approach they conduct a small case study. For this purpose, they utilize java projects as subject of investigation. For code clone detection this approach assign hash values calculated from a method name to each cloned segment and normalized tokens to each clone set as a unique identifier. Results indicate that method level clones (type1, type2 type3 or type4) are identified.

Table 2.7: Summary of research studies using Tree Based Approaches.

Sr.#	Reference No	Algorithm Used	Intermediate Representation	Machine Learning	Open Source software's	Clones Detected	Tool Support
1	[45]	Smith-Waterman	Variant of AST	N/A	JDK, Ant, Tomcat, ANTLR , dnsjava	Function	N/A
2	[46]	BFGS quasi-Newton method, MULTI-OBJECTIVE Genetic, NSGA-II	AST	Time Series Analysis ,NN	ArgoUML	1, 3	CloneDr
3	[47]	pattern recognition	AST	Pattern Recognition	N/A	1,2	N/A
4	[48]		AST, full binary tree	N/A	BigCloneBench	1,2,3,4	N/A

2.2.3. Tree Based Approaches

In tree-based clone detection techniques the program is parsed to parse tree or abstract syntax tree with the help of laxer or parser. After that similar subtrees are searched by using a tree matching approach. When it matches, the corresponding source code of similar subtrees is returned as clone class or clone pair. In **Table 2.7**, CCD based on these approaches is analyzed with the parameters given below.1) Algorithm Used evaluates, the algorithms utilized to measure similarity in these studies.2) Intermediate Representation shows the state in which source code is converted before clone detection. 3) Machine Learning describes the machine learning techniques utilized for clone detection 4) Open Source Software's indicate that whose datasets or source code used for CCD. 5) Clones Detected describes types of clones detected in these studies. 6) Tool Support describes whether the tool used or develop support the Tree-based approaches. The summary of these research studies is given in subsequent paragraphs.

In **Table 2.7,** Yang et al. [45] propose a CCD technique based on automated functions. Firstly, it creates AST from functions, transforms it into a new tree structure and then utilizes the Smith-Waterman algorithm to obtain similarity score between functions. The experiment is conducted by using five open source projects (JDK, Ant, Tomcat, ANTLR, dnsjava) and function level (1, 2, 3 or 4) clone are detected. Pati et al. [46] discuss a method for appropriate checking and predicting evaluation of clone numbers across various versions of open source software applications by comparing three models BP-NN, ARIMA and MOGA-NN. For this purpose, it

utilizes AST as an intermediate format, Multi-Objective Genetic Algorithm (MOGA) for optimizing two objective functions as a cost function, BFGS quasi-Newton method for training neural network and Time series to evaluate clone components. ArgoUML is applied for the implementation of the experiment and CloneDr used as a support tool. This method uncovers type1 and type3 clones. Chodarev et al. [47] proposed an algorithm for clone detection in the program source code. For this purpose, AST is utilized as an intermediate state and Pattern recognition algorithm is used to identify potential clones. It has the ability to detect type1 and type2 clones. Zeng et al. [48] propose a novel approach for fast code clone detection based on Weighted Recursive Autoencoders(RAE). For this purpose, they used BigCloneBench a benchmark dataset which consist of java code. This java code transforms into abstract syntax tree and then into full binary tree. Moreover, abstract syntax trees are analyzed with the help of weighted RAE, extract program features and encode the functions to vectors. The NSG algorithm is used for measuring similarity. Results indicate that type1, type2, type3 and type4 clones are detected.

2.2.4. Metric Based Approaches

In metric-based approaches, metrics are utilized to measure clones in software after the calculation from source code. For syntactic units such as function software or class, statement metrics are calculated and after that, comparison of these metrics values is performed. If two syntactic units have the same metric value, they can be considered as clone pair. For the calculation of the metric, this technique can also parse the source code to AST/PDG representation.

In **Table 2.8**, CCD based on metric-based approaches is analyzed with the following parameters.1) **Input Type/Intermediate State** shows that input taken by clone detection technique or intermediate format in which source code transform before clone detection. 2) **Similarity Measure** shows measuring of similarity for clone detection. 3) **Dataset** describes the datasets available in these studies on which clone detection is performed.4) **Machine Learning** evaluates the machine learning techniques or algorithms utilized for clone detection.5) **Clones Detected** describe types of clones identified in these studies. The summary of these research studies is given in subsequent paragraphs.

In **Table 2.8**, Tsunoda et al. [49] assess the differences in clone detection methods utilized in fault-prone module prediction. For this purpose, the source code is used as input, the dataset is collected from Lucene 2.4.0, an open source software, and Logistic regression is used to build

prediction models. Svajlenko and Roy [50] overviewed the concepts of CloneWorks, a near miss (type 3) clone detection tool by using IJaDataset, Jaccard similarity metric for clone detection. Sudhamani and Rangarajan [51] propose a method to detect duplicate clones. The experiment is conducted on dataset downloaded from fisourcecode. The source code is utilized as input, the similarity is measured by applying the self-defined formula and K-mean clustering is utilized for grouping the similar values where K=2. It is able to find all types (type1, type2, type3 and type4) of clones. In another work, Svajlenko and Roy [52] provide further details of CloneWorks, a clone detector by utilizing IJaDataset, source code as input and Jaccard similarity metric for clone detection. Results show that, it can identify type1, type2 and type3 clones. Haque et al. [53] develop a generic technique to detect code clones from different input source codes by dividing the code into a number of functions or modules. This approach is a combination of more approaches and methods. It has the ability to uncover all types of clones. Ragkhitwetsagul et al. [54] present an image based clone detection approach and a tool named Vincent. It applies java source files as a dataset, transforms it into PNG image as an intermediate state and then utilizes Jaccard similarity for clone detection. Results indicate that type1, type2 and type3 clones are identified. Sudhamani and Rangarajan [55] address structure similarity detection using the structure of control statements. For this purpose, C/C++, java files are used as dataset and self-defined formula used for similarity computation. This method can efficiently uncover structurally similar (type1, type2, type3, or type4) clones.

Yu et al. [56] propose a technique for code clone detection based on bytecode sequence alignment. For this purpose, java code transforms into bytecode and Smith Waterman algorithm is used to align bytecode sequences for precise matching. They calculate the similarity of two code fragments by measuring their cosine distance. This approach can detect type1, type 2, type3 and some type4 clones. Sudhamani and Rangarajan [57] propose a technique for detection of code similarity through program statement and control statements. To evaluate the performance of proposed approach the experiment is conducted on 93 lab programs designed by students and source code of 4 C projects of Billon's benchmark. Furthermore, clones are detected by clustering similar values. The details about different types of clones are not given.

Table 2.8: Summary of research studies using Metric Based Approaches.

Sr.#	Reference No	Input Type/ Intermediate State	Similarity Measure	Dataset	Machine Learning	Clones Detected
1	[49]	Source Code	N/A	From Lucene	Logistic Regression	N/A
2	[50]	Source Code	Jaccard	IJaDataset	N/A	1,2, 3
3	[51]	Source Code	Self-defined Formula	From fisourcecode	K-mean clustering	1,2,3,4
4	[52]	Source Code	Jaccard	IJaDataset	N/A	1,2, 3
5	[53]	Source Code	N/A	N/A	N/A	1,2,3,4
6	[54]	PNG Image	Jaccard	Java source code Files	N/A	1, 2,3
7	[55]	Source Code	Self-defined Formula	C/C++ Java files	N/A	structural
8	[56]	Byte code	Cosine distance	Java	N/A	1,2,3,4
9	[57]	Program statements Control Statements	N/A	C	Clustering	N/A

2.2.5. Semantic Approaches

In these techniques, the program is represented as a program dependency graph (PDG). Approaches that depend on program dependency graph goes one-step further to obtain high abstraction of source code representation than others because it considers semantic information of the source. Program dependency graph carries control flow and data flow information and hence contain semantic information. Once a set of PDGs is obtained, the isomorphic subgraph matching algorithm is applied for finding similar subgraphs which are returned as clones.

In **Table 2.9** CCD based on semantic approaches is analyzed with the help of following parameters.1) **Algorithm Used** shows that the algorithms utilized for identification of clones.2) **Similarity Measure** indicates measuring of similarity for clone detection. 3) **Language** represents the language of the source code, which is transformed into PDG. 4) **PDG Constructor** describes a framework or anything that helps in the construction of PDG from source code. 5) **Clone Type Detected** evaluates types of clones detected in these studies. Summary of these studies is given in subsequent paragraphs.

In **Table 2.9**, Wang et al. [58] present CCSharp: An efficient three-phase clone detector using modified PDGs. It applies Frama-C2 to generate program dependency graphs of source code in C language. It utilizes Vector filtering algorithm to exclude the PDG pairs, which are not likely to be cloned. Euclidean distance is used to measure the numerical similarity and Levenshtein Distance is utilized to measure string similarity. As this tool is based on PDG based approach, so we can suppose it has the ability to detect type 4 clones. Sabi et al. [59] examine how clone detection result changes by rearranging the program statements by using PDGs. For this purpose, the Java source code is used. Results show that type1 and type2 clones are identified. Crussell et al. [60] propose AnDarwin, a tool for finding applications with the similar code on large scale by utilizing WALA to generate program dependency graphs of source code in C language, LSH algorithm for finding an approximate nearest neighbour in a large number of vectors and Min-Hash algorithm to measure partial or full app similarity. Sargsyan et al. [61] propose an algorithm for scalable and accurate clone detection. For this reason, PDG is constructed by a compilation of C program files, LLVM is used as compilation infrastructure and Isomorphism algorithm is used for similarity measure. The proposed algorithm can identify type 4 clones. Similarly, Hu et al. [62] present another algorithm by utilizing java source code to identify new clone relations from the clone pair results of PDG base detection. For this purpose, the ASM algorithm is used and type 4 clones are identified.

Kamalpriya and Singh [63] propose a semantic-based approach to find functions of binary clone and implement this approach in a prototype system named CACOMPARE. The experiment is conducted by using the binary code in assembly language, IDA Pro dissemble this code and extract CFGs, Min-Hash algorithm to quickly estimate the Jaccard index and LCS algorithm for similarity score computation. The result indicates that type 4 clones are detected. Avetisyan et al. [64] present a framework for CCD that is based on LLVM. It utilizes the source code written in C language and LLVM for the transformation of bytecode into PDGs. It uncovers type1, type2, type3 and type4 clones.

Table 2.9: Summary of research studies using Semantic Approaches.

Sr.#	Reference No	PDG Constructor	Language	Algorithm Used	Similarity Measure	Clone Type Detected
1	[58]	Frama-C2	C	Vector filtering	Euclidean Distance, Levenshtein Distance	4
2	[59]	N/A	Java	N/A	N/A	1,2
3	[60]	WALA	Java	LSH	Min-Hash	4
4	[61]	LLVM	C	Fast Checking	Isomorphism	4
5	[62]	IDA Pro	Assembly	LCS	Min-Hash	4
6	[63]	N/A	Java	ASM	N/A	4
7	[64]	LLVM	C	N/A	N/A	1,2,3,4

2.2.6. Hybrid Approaches

The combination of two or more CCD approaches (Textual, Lexical, Syntactic or Semantic) is called a hybrid approach. The hybrid approach holds better results than the normal one [63]. CCD based on hybrid approaches is analyzed in **Table 2.10** with the help of following parameters.1) **Hybrid** shows the combination of clone detection approaches used as a hybrid. 2) **Dataset** indicates the dataset available in the form of source code which is converted into different states for clone detection.3) **Transformation** describes the source code undergos in different forms for clone detection. 4) **Algorithm Used** shows the availability of algorithms for similarity measure or clone detection. 5) **Clone Type Detected** shows that types of clones detected in these studies. The summary is given below.

In **Table 2.10**, Singh [65] focuses on enhancements in the CCD algorithm by using a hybrid approach, that is a combination of metric based approach and PDG based approach. The dataset used by this approach is Java source code, which is transformed into AST and PDG. It can identify type1, type2 and type3 clones. Misu and Sakib [66] develop an interface driven CCD approach (IDCCD) by combining token based and metric-based approaches. For this purpose, IJaDataset is used that transforms into regularized tokens and ASTs. It has the ability to find type1, type2 and type3 clones. Sheneamer and Kalita [67] propose an efficient metric based approach for clone

detection and it extracts features from ASTs and PDGs. It utilizes IJaDataset 2.0(Java code) that undergoes AST and PDG transformation, and Rotation Forest, Random Forest, Xgboost algorithms that can detect clones automatically. This approach can find type1, type2, and type3 and type4 clones. Vislavski et al. [68] describe LICCA, a tool for cross-language clone detection that is a combination of token based, AST based and metric-based approaches and uncovers type1, type2 and type3 clones. This tool utilizes Java, C, JavaScript, Scheme and Modula-2 code as a dataset, eCT representation for AST based detection and a variant of LCS algorithm for token based detection. Misu et al. [69] describe an exploratory study on interface similarity in code clones. For this purpose, token-based and text-based tools are used and Java source code undergoes AST transformation. Results indicate that type1, type2 and type3 clones are identified. Akram et al. [70] develop Droid CC a clone detection approach, by combining text-based and token based approaches for android applications. The dataset utilized by this approach is java code that transforms into regularized tokens. The MD5 Hashing algorithm is used to get hash values against each chunk. This approach can detect type1, type2 and type3 clones.

Sheneamer et al. [71] introduce a framework for obfuscated and semantic clones by using machine learning. It is a combination of semantic and tree-based techniques and Java code is used as a dataset that transforms into BDG, AST and PDG. In order to train and test the model ensemble approach (majority voting) among ten classifiers (Naïve Bayes, IBK, SVM, Logit Boost, Ran-dom Subspace, Random Committee Rotation Forest Random Forest J48) is utilized. This framework can detect type1, type2, type3 and type4 clones. Matsushita et al. [72] present an algorithm that detects clones with gaps by using the combination of token based and AST based approaches. ML programs used as a dataset that transforms into regularized tokens and ASTs. The algorithm can find type1, type2 and type3 clones. Similarly, Tekchandani et al. [73] present another algorithm that is a hybrid of token based, AST based and semantic approaches for IoT applications by using Java source code that transforms into Tokens and ASTs. It can identify type 4 clones. Uemura et al. [74] propose a method CCD in Verilog HDL by combining token based and metric-based techniques. The dataset employed by this method consists of HDL code which is transformed into C++ code and then into tokens. It can detect type1 and type2 clones. Nasirloo and Azimzadeh [75] present a method for semantic (type 4) CCD using AMSs and PDGs. It applies C source files as a dataset that transforms into tokens PDGs and AMSs before clones are detected. Singh and Sharma [76] present a hybrid approach (text-based + metric Based) to detect file level clones for high-level

cloning by applying dataset that consist of C, C#, Java and text files. It can detect structural (1, 2, 3 or 4) clones. Sheneamer and Kalita [77] present hybrid CCD technique using fine-grained and coarse-grained techniques. It utilizes the combination of lexical and metric-based approaches, Murakami's (java files) dataset that transforms into regularized tokens and uncovers type1, type2 and type3 clones.

White et al. [78] present a technique for CCD based on deep learning of code fragments by combining lexical, tree-based and metric-based techniques. This technique utilizes RtNN as deep learning algorithm and Java source code as a dataset, which transforms into tokens and ASTs. ASTs then transform into a full binary tree. It can detect type 3 clones. Ragkhitwetsagul et al. [79] compare the code similarity analyzers. For this purpose, they utilized java source code and transforms it into bytecode, which is utilized for similarity or clone detection. For clone detection bytecode further transforms into tokens or ASTs by using different clone detection tools (simian, NICAD, CCFinderX, iclones, Deckard) supported by clone detection techniques (text-based, token-based, tree-based). Results indicate that these clone detection tools and techniques perform better than general similarity measures. Ghofrani et al. [80] introduce a framework for clone detection using a deep neural network as a machine learning algorithm and hybrid (token based and metric-based) technique as a CCD technique. Regularized tokens utilized as an intermediate format and type 4 clones are identified.

Liu et al. [81] present a dynamic parameter based sequence alignment algorithm for detecting large gap clones including first three types. It is a combination of token based and metric based approaches, java source code used as dataset which is transform into tokens. These tokens then normalized some rules and generate fingerprints by using MD5. The sequence alignment is Smith-Waterman algorithm based. In the detection phase, dynamic parameter acquisition strategy is used to optimize the key parameters in the Smith-Waterman algorithm. The similarity between two code fragments is measured by using self-defined formula. Yu et al. [82] propose an approach for detection of semantic code clones through tree based convolution. For this purpose, they capture both lexical information of code segment from code token and structural information from its AST, utilize two neural network for the detection of code clones and similarity is measured by calculating cosine similarity between two vectors. Datasets utilized by this approach are BigCloneBench and OJClone. In BigCloneBench type1, type2, type3 and type4 clones are detected while in OJClone results are not clearly stated.

Table 2.10: Summary of research studies using Hybrid Approaches.

Sr.#	Reference No	Hybrid	Dataset	Transformation	Algorithm Used	Clone Type Detected
1	[65]	PDG Based + Metric Based	Java Code	AST + PDG	N/A	1,2,3
2	[66]	Token Based + Metric Based	IJDataset2.0 (Java code)	AST+ Tokens	N/A	1,2,3
3	[67]	AST Based + PDG Based	IJDataset2.0 (Java code)	AST+PDG	Rotation Forest, Random Forest, Xgboost	1,2, 3,4
4	[68]	Token Based+ Tree Based + Metric Based	Java ,JavaScript , C , Modula-2 , Scheme	eCT	LCS	1,2,3
5	[69]	Token Based + Tree Based	Java code	AST	N/A	1,2,3
6	[70]	Textual + Token Based	Java code	Tokens	MD5 Hashing	1,2,3
7	[71]	AST + Semantic	Java code	BDG , AST , PDG	Naïve Bayes SVM (IBK) Logit Boost Random Committee Ran- dom Subspace Rotation Forest Random Forest J48	1,2,3,4
8	[72]	Token Based + AST based	ML programs	Regularized tokens ,AST	N/A	1,2,3
9	[73]	Token Based+ AST Based + Semantic	Java Code	Tokens ,AST	N/A	4
10	[74]	Token based + Metric Based	HDL code	C++ ,Tokens	N/A	1,2
11	[75]	Token Based + PDG Based	C code	Tokens, PDG, AMS	N/A	4
12	[76]	Text Based + Metric Based	C , C# , Java ,Text files	N/A	N/A	Structural
13	[77]	Lexical + Metric Based	Murakami's(Java files)	Regularized Tokens	N/A	1,2,3
14	[78]	Lexical + Tree Based+ Metric Based	Java Code	AST	RtNN	3
		Text Based +		Bytecode,		

15	[79]	Token Based + Tree Based	Java Code files	Tokens, AST	N/A	N/A
16	[80]	Token Based + Metric Based	N/A	Tokens	DNN	4
17	[81]	Token-Based +Metric Based	Java	Tokens	Smith-Waterman	1,2,3
18	[82]	Token-Based+ Tree Based	BigCloneBench, OJClone	Tokens, AST	NN	1,2,3,4

2.2.7. Open Source Subject Systems

We overall identify 67 open source subject systems whose datasets or source code used for CCD. These are summarized in **Table 2.11** with the following parameters: 1) **Subject System** indicates that the name of the open source subject system whose source code utilized for CCD. 2) **Language** shows that the implementation language of these open source subject systems. 3) **License Type** represents the type of license under which they release. 4) **Reference** of the paper is provided for further detail.

We hope that **Table 2.11** may help researchers in the selection of most frequently used open source subject systems as a benchmark for evaluation and empirical studies. Other than these subject systems, IJaDataset which is a benchmark dataset consist of java code repository is also extensively used for code clone detection.

Table 2.11: Summary of open source subject systems used in selected studies.

Sr.#	Subject System	Language	License Type	References
1	JDK	Java	MPL	[45][78][38][40]
2	Ant	Java	GPL	[45][24][38][63][77][78][56]
3	Tomcat	Java	GPL	[45][54]
4	ANTLR	Java	BSD	[45][65][42][78][21][74]
5	JEdit	Java	GPL	[65][44]
6	Eclipse	Java	EPL	[39]
7	Qpid	Java	ASL	[59]
8	Subversion	Java	GPL	[59]
9	Wookie	Java	GPL	[59]

10	Hibernate	Java	GPL	[78]
11	ArgoUML	Java	EPL	[46][78]
12	Swing	Java	GPL	[74][77][56]
13	Junit	Java	EPL	[21][54][44]
14	JfreeChart	Java	LGPL	[21][54]
15	Hadoop	Java	GPL,AGPL	[39]
16	Neo4j	Java	GPL,AGPL	[39]
17	EIRC	Java	GPL	[71]
18	Netbeans	Java	GPL	[24][77][56]
19	jdtcore	Java	EPL	[24][77][56]
20	JhotDraw	Java	LGPL	[78][34]
21	OpenNLP	Java	GPL	[38][79]
22	Maven	Java	GPL	[38][79]
23	RxJava	Java	N/A	[74]
24	Okhttp	Java	GPL	[74][44]
25	React-native	Java	GPL	[74]
26	Wget	C	GPL	[71]
27	Postgresql	C	GPL	[38]
28	LLVM	C/C++	GPL	[61][57]
29	Ffmpeg	C	LGPL	[32]
30	Firefox	C/C++ JavaScript	MPL	[32][61]
31	OpenSSL	C	BSD	[25][61][32]
32	HTTPD	C/C++	GPL	[25][32][22]
33	Linux	C	GPL	[25][32][61][38][74][64][40]
34	Mono	C#	LGPL	[40]
35	MonoDevelop	C#	LGPL	[40]
36	Webogram	Python	GPL	[26]
37	DNSjava	Java	BSD	[45][78]
38	jUDDI	Java	GPL	[39]
39	Gora	Java	GPL	[59]

40	Cook	C	N/A	[38][57]
41	netdata	Java	GPL	[74]
42	Redis	C	BSD	[75][38]
43	FREECOL	Java	GPL	[65]
44	Commons Lang	Java	GPL	[55][40][44]
45	Wink	Java	GPL	[39][55]
46	Lucene	Java	GPL	[49]
47	Google Android	Java C/C++ Python	GPL	[25]
48	Codeaurora Android	Java, C /C++	GPL	[25]
49	Google Chromium	C/C++ Java,Python	MPL,GPL, LGPL	[25]
50	Ubuntu-Trusty	Java C/C++ Python	GPL	[25]
51	Roller	Java	GPL	[39]
52	OpenOffice	Java	LGPL	[39]
53	OODT	Java	GPL	[39]
54	Onami	Java	GPL	[39]
55	JSPWiki	Java	GPL	[39]
56	Forrest	Java	GPL	[39][55]
57	Any23	Java	GPL	[39][59]
58	BVal	Java	GPL	[59]
59	Flume	Java	GPL	[59]
60	Giraph	Java	GPL	[59]
61	cTAKES	Java	GPL	[39]
62	FOP	Java	GPL	[73]
63	iTrust	C\C++	N/A	[33]
64	druid	Java	GPL	[44]
65	SNNS	C	GNU LGPL	[57]
66	Weltab	C	N/A	[57]
67	SLP	C	N/A	[57]

2.3. RESEARCH GAPS

The preceding review of the existing literature indicates that lots of techniques and tools are developed for the detection of code clones. Most of these techniques utilize java source code to identify clones. Consequently, source code of other programming languages should be target to examine the efficiency of these techniques.

It is analyzed that most of these techniques are unable for the detection of semantic or type 4 clones, because these clones are semantically same but structurally different. Few tools or techniques that can detect these clones utilize traditional methods which can weakly detect type 4 clones [9]. From literature we find few (3 or 4) studies that tried their best to detect all types of clones including with good results (accuracy, execution) but their capabilities are limited to java code only [48][67][71][82] because parsers or compilers used by them are restricted to java code. However, current approaches are incompetent to find semantic clones along with other (type 1, type 2 and type 3) three types of clones with good results in programing languages (e.g. C/C++).

Moreover, from literature it is examined that source code of open source software systems is used for CCD. Similar to open source systems clones can also exist in commercial software systems. Therefore, in future commercial software systems should be target to check the validity of these approaches on commercial level.

Chapter 3

Proposed Methodology

CHAPTER 3: PROPOSED METHODOLOGY

This chapter consists of the proposed solution of identified problem. In this chapter **Section 3** consists of code clone detection framework. This section further divided into sub sections, **Section 3.1** consists of AST generation, **Section 3.2** tells about feature extraction. Similarly, **Section 3.3** discuss about the fusion of code features. Furthermore, **Section 3.4** discuss the clone detection scheme**.**

3. CODE CLONE DETECTION FRAMEWORK

Basically code clone detection is considered as an analysis of pairwise similarity problem. According to this problem, if a fragment of code is syntactically similar with given reference code then these two code fragments are syntactic clones, or if the fragment of code semantically similar to the reference code then these two clones are semantic clones. However, for training and testing machine learning mostly examines individual samples. To compare semantic or syntactic similarity between two code fragments, we can extract the related aspects of code segments by viewing their associated structure or their selected parts. In literature of machine learning these aspects known as *features*. For pairwise detection of code clones by applying machine learning, we utilize features of both reference and target code segments.

Pairwise Learning (Definition): We provide a set of N pairs of training samples, depending on mutual similarity, each sample which contain a pair of code segments labelled with clone type. The classification model act as a mapping function f: A $\rightarrow$ B, where A is pair of code segments which is unknown and B is possible type of clone predicted by a classification model. We represent the training sample as feature vectors. These feature vectors are represented as features $(S_i, S_j) = (f_1, f_2, \dots, f_n, C_k)$ of n size. It is built by associating the features of two different code fragments (S_i, S_j) and class C_k. By combining C_k with (S_i, S_j)sample of training matrix of size N(n+1) is formed.

Similarity between two feature vectors is measured to measure the similarity between two code segments. The related features for a pair of code segment comes from various sources. One of them is from abstract syntax tree representation. In our research work we use features comes from abstract syntax trees.

In next step we explain comprehensively different features of code segments we utilized for code clone detection. The overview diagram is Shown in **Figure 3. *1*.**

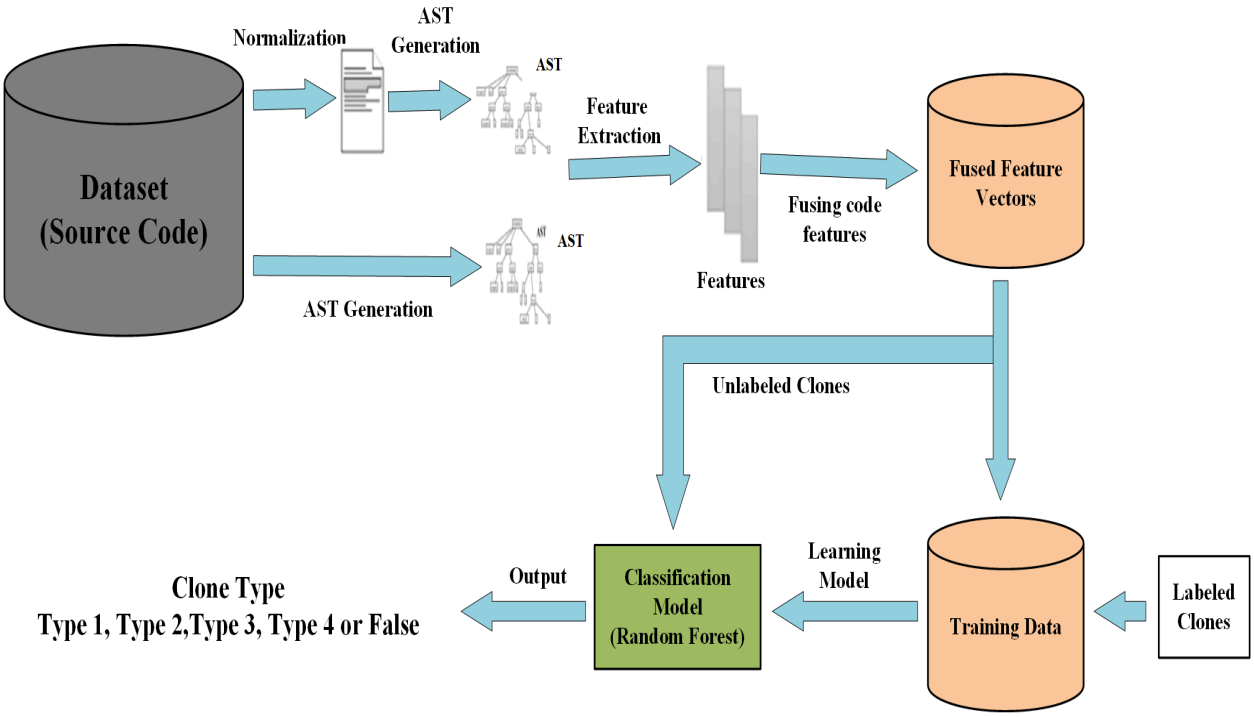


Figure 3. 1: Overview Diagram

3.1. AST Generation

To measure the similarity between two code segments we parse these segments into their constitutional segments in the form of Abstract Syntax Trees(ASTs) with the help of AST generator or parser [83]. Abstract syntax tree consists of nodes and edges. Every tree node shows the programming construct present in the provided program and leaf nodes of the tree consist of variables. A general example of AST is shown in **Figure 3.2**.

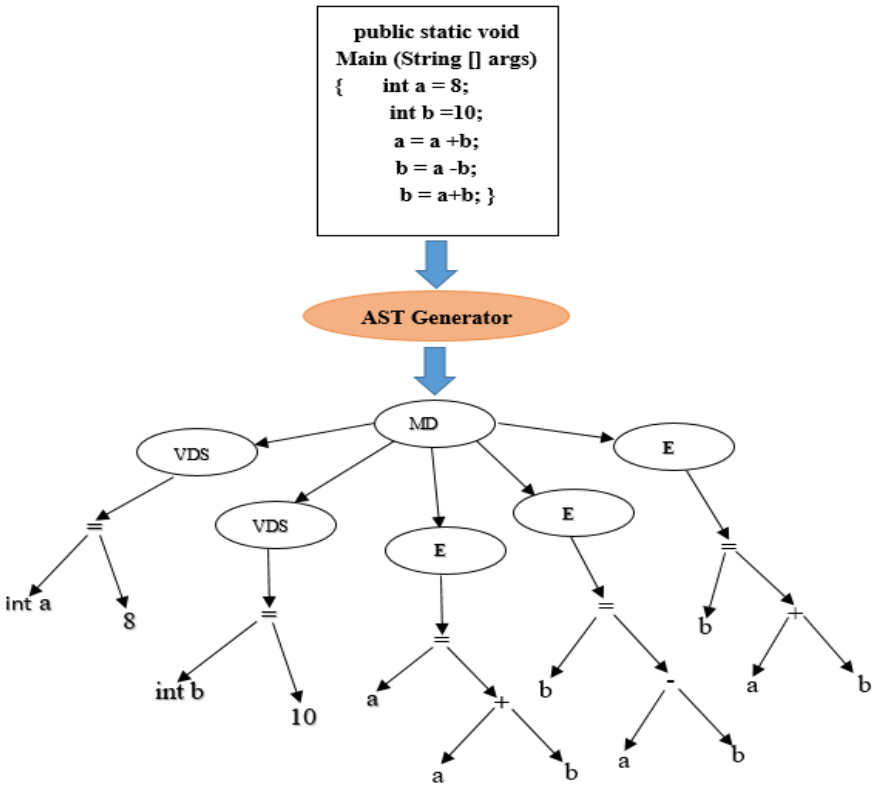


Figure 3.2: AST generated from code segment

In **Figure 3.2** MD: represents method declaration, VDS: variable declaration statement and E: expression.

We utilize a tool, named frama-c [1]for generating ASTs of source code. This tool is used to generate ASTs of C or C++ code. However, frama-c build its internal ASTs after parsing the source code.

3.2. Features Extraction

We extract features from normalized and original ASTs by using metrics[2] plugin of frama-c.

3.2.1. Normalization

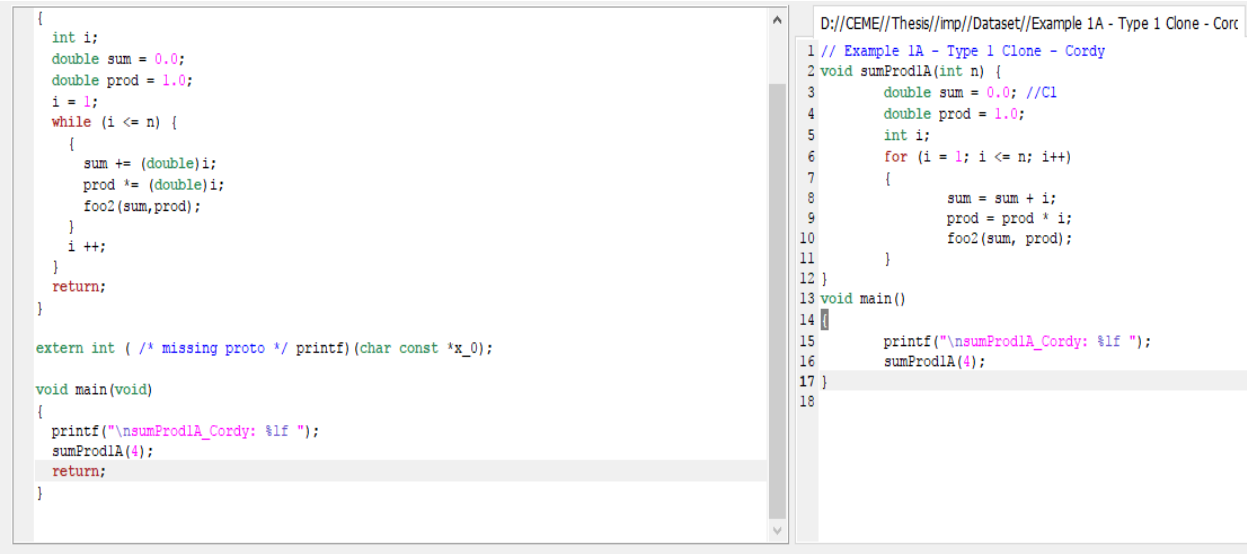
The process of normalization simplifies the source code by making some syntactic changes. For example:

- Normalization process convert all kinds of loops into while loop.

[1] https://frama-c.com/
[2] https://frama-c.com/metrics.html

- This process introduces return statements which are missing.
- This process adds some temporary variables.

The changes in source code during normalization process made by frama-c are shown in **Figure 3.3.**



```
{
  int i;
  double sum = 0.0;
  double prod = 1.0;
  i = 1;
  while (i <= n) {
    {
      sum += (double)i;
      prod *= (double)i;
      foo2(sum,prod);
    }
    i ++;
  }
  return;
}

extern int ( /* missing proto */ printf)(char const *x_0);

void main(void)
{
  printf("\nsumProd1A_Cordy: %lf ");
  sumProd1A(4);
  return;
}
```

D://CEME//Thesis//imp//Dataset//Example 1A - Type 1 Clone - Corc

```
1 // Example 1A - Type 1 Clone - Cordy
2 void sumProd1A(int n) {
3         double sum = 0.0; //C1
4         double prod = 1.0;
5         int i;
6         for (i = 1; i <= n; i++)
7         {
8                 sum = sum + i;
9                 prod = prod * i;
10                foo2(sum, prod);
11        }
12 }
13 void main()
14 {
15        printf("\nsumProd1A_Cordy: %lf ");
16        sumProd1A(4);
17 }
18
```

Figure 3.3: Source code normalization by Frama-c

As this process does not affect the semantic contents of the source code but its syntactic structure may change. Accordingly, it can affect the features extracted from source code. Therefore, we extract features from both normalized and original ASTs to get better results. Features or metrics which extracted from normalized ASTs are cyclomatic features and features from original ASTs include halstead features and cyclomatic features. We utilize cyclomatic features of normalized ASTs and halstead features from original ASTs. Cyclomatic features of original ASTs are not utilized as these features do not add some improvement in the results rather they increase the redundancy.

3.2.2. **Features extracted from Normalized ASTs** (Cyclomatic Features)

- SLOC: Source line of code
- Decision point: Conditional statements e.g. if, switch cases, loops
- Global variables: No. of global variables appeared in code
- If: No. of if statements occur in a piece of code

- Loop: No. of loops, basically while loops as all loops are converted into while loop after normalization
- Goto: No. of goto statements occurs in a piece of code
- Assignment: No. of assignments in a code segment
- Exit point: No. of return statements occurs in code fragment
- Function: No. of function declared
- Function call: No. of function calls in code segment
- Pointer dereferencing: Getting the no. value that is stored in the memory location pointed by the pointer
- Cyclomatic complexity: $C = \pi - s + 2$ (π no. of decision points and s no. of exit points)

3.2.3. **Features extracted from original ASTs** (Helstead Features)

- Total operators: N_1
- Distinct operators: η_1
- Total operands: N_2.
- Distinct operands: η_2
- Vocabulary size: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$
- Program level: $\tilde{N} = \eta_1 * \log_2 \eta_1 + \eta_2 * \log_2 \eta_2$
- Program volume: $V = N * \log_2 \eta$
- Difficulty level: $D = \left(\frac{\eta_1}{2}\right) * N_2/\eta_2$
- Effort: $E = D * V$
- Time to implement: $T = E/18$
- Bugs delivered: $B = E^{2/3}/3000$

All these feature/metrics are extracted automatically from ASTs of source code by using metrics plugin of frama-c in frama-c gui. The details of total no. of features are given in **Table 3.1**.

Table 3.1: Summary of total no. of features extracted from AST.

Total no. of features	From normalized AST	From Original AST
24	12	12

3.3. Fusion of Code Features

In this step the extracted feature vectors from a pair of reference and target code are combined to generate training dataset as presented in **Eq. (3.1)**.

$$[S_i^n] \approx \left[f_{i1} \dots f_{in_n} \mid f_{i1} \dots f_{in_o}\right] \tag{3.1}$$

The above equation shows that different types of features denoted with different notations: n for features extracted from normalized AST and o for features extracted from original AST. For clear separation vertical line is used between different groups of features.

To measure the similarity between two code fragments we fuse the sequence features of these two code fragments. Although, there are two types of features, features of normalized AST and features of original AST in the description of code segment. We can rewrite the equation to simplify the notation without distinguishing among the types of features as presented in **Eq. (***3.2* and **Eq.(***3.3*.

$$[S_i^n] \approx features\ (S_i) = \ [f_{i1} \dots f_{in}] \tag{3.2}$$

where $n = n_n + n_0$

$$\left[S_j^n\right] \approx features\left(S_j\right) = \left[f_{j1} \dots f_{jn}\right] \tag{3.3}$$

For feature extraction and labeling of the class of the feature vector as true clone, we utilize known pairs of cloned fragments.

Given two code segments S_i and S_j and their corresponding class label C, the merged feature vector *features* (S_i, S_j) can be presented as fused feature vector.

From literature we find three ways for fusing the features of reference code segment and target code segment to measure the similarity between them.

These combination ways include

- Linear Combination
- Multiplicative Combination
- Distance Combination

We try all these combinations on our both datasets. The results of all these combination ways presented in **Table 3.2**.

Table 3.2 Summary of results using different ways of combining features.

Sr.#	Combination Type	Dataset	Results
1	Linear Combination	Krawitz	97.5%
		Roy et al.	92.6%
2	Multiplicative Combination	Krawitz	80%
		Roy et al.	81.96%
3	Distance Combination	Krawitz	76%
		Roy et al.	79.8%

From **Table 3.2** we find that linear Combination gives better results than multiplicative or distance combinations. Therefore, we utilized linear combination in our framework.

By utilizing linear combination, two feature vectors are fused, as explained below.

3.3.1. Linear Combination

In this combination two feature vectors are simply linked. This linkage provides the fused feature vector of size 2n. This combination is given below.

$$features\left(S_i, S_j\right) = \left[f_{i1} \ldots f_{in}, f_{j1} \ldots f_{jn}, C\right] \quad (3.4)$$

3.4. A Code Clone Detection Scheme

A machine learning approach is utilized for code clone detection. The proposed scheme consists of two stages, first is training and the second is testing like any other machine learning approach. In training phase, from given corpus known pairs of code clones are utilized. In first step, normalization on source code is performed. Next source code is parsed to generate ASTs,

there are two kinds of ASTs normalized AST and original AST. These are discussed in detail **section 3.1**. After that, features are generated from both normalized AST and original AST and fuse feature vectors of two code fragments (reference and target) by utilizing **Eq.**(*3.4*). Furthermore, based on the type of clone (type1, type2, type3 or type4) the feature vector is labeled with a class label.

For creating training dataset for classification model, every aforementioned step is repeated for all possible pairs of code fragments. In order to find possible type of clone in unlabeled code segments, the same sequence of steps is performed to create feature vector of the two given code fragments (reference and target) and pass it through the classification model to predict the possible type of clone (type 1, type 2, type 3 or type 4). The classification model we utilized in our approach is random forest with the belief of attaining good results.

Chapter 4

Experimentation and Results

CHAPTER 4: EXPERIMENTATION AND RESULTS

In this chapter the experimentation details and results of this research work are provided. In this chapter **Section 4** consists of experimental evaluation of proposed framework. This section further divided into sub sections, **Section 0** consists of design assessment, **Section** Error! R eference source not found. tells about datasets we utilized in our work and performance of our framework in terms of accuracy on these datasets. In **Section 4.3** our proposed framework is compared with other code clone detection approaches in terms of accuracy. Moreover, **Section 4.4** discuss about the extended experiments we performed to get more accurate results. **Section 4.5** consists of classification model we utilized in our framework. **Section 4.6** discuss about the performance of our approach in terms of execution time. Furthermore, **Section 4.7** discuss the performance of proposed framework with different training and testing samples.

4. EXPERIMENTAL EVALUATION

It must be noted that each fragment of code need to be parsed/compiled successfully by the parser/compiler in order to work this framework. For parsing source code, our framework utilizes parser of a tool, named frama-c. The performance of our approach is evaluated in the light of open source code clone datasets and compared with popular code clone detection approaches. Furthermore, in our experiment we utilize examples of only C source code corpus. We perform this experiment using Matlab 2016 (a). However, this framework can be extended to any high level programming language because this is generic in nature. Our main objective is to increase the accuracy of type 4 or semantic clones in C source code while not compromising the accuracies of other three types of clone. For training and testing we utilize random forest which is a supervised learning based classification model and compare the capability of our approach with popular code clone detection approaches. The details of whole process are provided in the subsequent sections.

4.1. Design Assessment

For training we utilize code fragments whom class labels are known. Afterwards, we extract features as discussed in **section 3.2** and fuse them by utilizing the scheme presented in **section 3.3**. Then we utilize classification model for training to attain higher accuracy. For the prediction of unknown clones same steps are repeated.

4.2. Datasets

We utilize two code clone datasets for training and testing. One is from Krawitz [84] and other from Roy et al [9]. Both these datasets consist of C source code. The granularity level of both these datasets are at functional level. Krawitz [84] dataset is a source code which he utilized to check the capabilities of his approach. Roy et al [9] dataset is basically a taxonomy which they made to evaluate the performance of various code clone detection techniques and tools. This taxonomy is not simply a guesswork, it is derived from a huge amount of work published on clone types [89] [90], clone definitions [83][85][86][87][88], clone taxonomies[91][92], developers copy paste activates and other empirical studies[93]. This taxonomy is validated by studies the copy paste patterns of function clones [94]. The details about datasets are given in **Table 4. 1**.

Table 4. 1: Detail about Datasets.

Dataset	Total Pairs	TYPE 1	TYPE 2	TYPE 3	TYPE 4
Krawitz[84]	9	2	2	2	3
Roy et al.[9]	16	3	4	5	4

As these datasets are small and partially labeled. They labeled only with respect to original piece of code and consists of small no. of combinations, so we use leave-one-out cross validation for training and testing of these datasets. The results of this validation are shown in **Figure 4.1**.

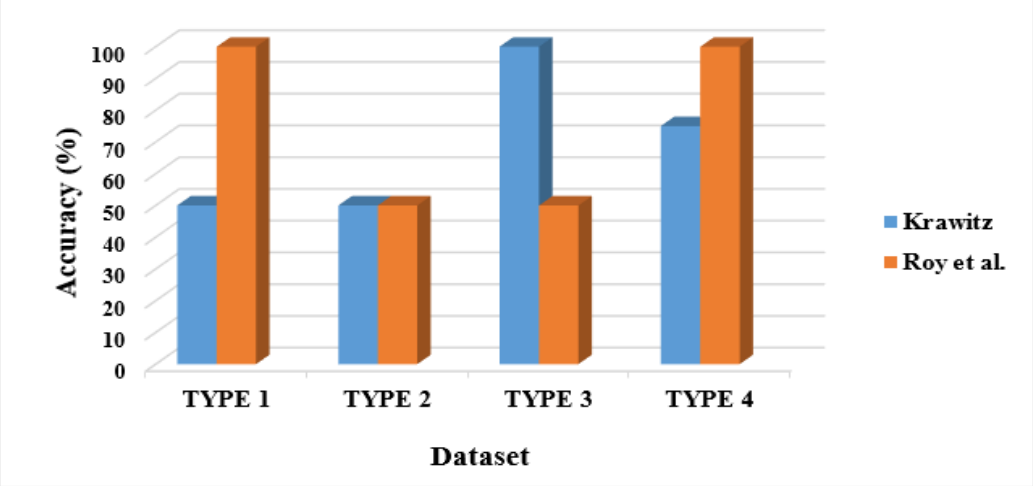


Figure 4.1: Detection Rate using leave-one out cross validation

4.3. Performance Comparison

The results of our framework (**Figure 4.1**) are compared with popular and state of the art code clone detection approaches that are used in several recent studies**.** For this purpose, we compare the modern code clone detection methods by utilizing the results reported on Roy et al. taxonomy. We compare our framework with various code clone detection approaches: Duploc, Basic NICAD, Full NICAD, Simian, CCFinder(X), iClones, CP-Miner, Dup, CloneDr, Deckard, Duplix, Gabel. Previous research addresses [9] a range of accuracy for detecting type 1, type 2, type 3 and type 4 clones only. The reason is that the datasets accessible to them are lacked false clone examples. Therefore, we conduct an analysis of detectors for their capabilities to detect type 1, type 2, type 3 and type 4 clones by utilizing different terms used by the prior researches. These terms are explained in **Table 4.2**.

Table 4.2:Terms utilized to define accuracy and their description

Term	**Explanation**
Very well	Detectors detects clones with very good accuracy(e.g,100%).
Well	Detects the clones but may returns some false Positive. May miss some of the clones.
Medium	Detects the clones but may returns many false positive (about 50% for example).
Low	Detects the clones with many false positive. Moreover, it is possible to miss lots of the similar clones. (low accuracy)
Probably can	Detect clones with lots of false positive and miss few code clones. (very low accuracy).
Probably cannot	We cannot found any empirical study that the detector is adequate to detect code clones.
Cannot	There is no any sort of evidence or empirical study that detector is competent to detect clones.

The comparison of our proposed framework with other popular code clone detection approaches in terms of accuracy is presented in **Table 4.3**. Results indicate that the proposed framework is very good in the detection of semantic or type 4 clones.

Our primary objective is to improve the accuracy of semantic or type 4 clones while not compromising the accuracy of other three types of clone. Results show that our approach is superior in the detection of Type4 clones and comparable in the detection of Type1 clones with other approaches. However, in the detection of Type 2 and Type 3 clones our approach does not give acceptable results. The reason is that the no. of combinations are small, for example there are two Type 3 clones, in one addition (adding some statement) occurs and in other deletion (deleting some statement) occurs after copy pasting. So if we train data with piece of code in which addition occurs and test data with code fragment in which deletion occurs then we cannot get desired results. Because the data we trained is different from the data we are testing. The similar kind of situation happens with Type2 clones. Therefore, we perform some extended work to get the desired results.

Table 4.3: Comparison of our framework with other clone detection approaches.

Clone detectors	**TYPE1**	**TYPE2**	**TYPE3**	**TYPE4**
Duploc [9]	well	cannot	Medium	cannot
Basic NICAD[9]	Very well	Well	Probably can	Probably can
Full NICAD[9]	Very well	Well	Well	Probably can
Simian[9]	Very well	Well	Cannot	Cannot
CCFinder(X)[9]	Well	Well	Cannot	Cannot
iClones[9]	Very Well	Well	Cannot	Cannot
CP-Miner[9]	Well	Well	Medium	Cannot
Dup[9]	Well	Well	Cannot	Cannot
CloneDr[9]	Very Well	Well	Medium	Cannot
Deckard[9]	Well	Well	Medium	Cannot
Duplix[9]	Well	Well	Medium	Low
Kontogiannis[9]	Well	Medium	Low	Medium
Gabel[9]	Well	Medium	Medium	Medium
Our Approach	Very Well	Medium	Medium	Well

4.4. Extended Experiments

Datasets we utilized in our research are partially labeled, they labeled only with respect to original piece of code as discussed in **section 4.2**. There is also a relationship between other pairs of code because all are occurring as a result of copy pasting. For example, Type 1 clone of original piece of code and Type 2 of original piece of code are also some kind of clones of each other, similarly Type 2 of original is some type of clone to Type 3 of original and so on. Therefore, we make each possible combination and manually labeled the unlabeled code fragments. We labeled these code fragments by deeply analyzing huge amount of work published on clone types [89] [90], clone definitions [83][85][86][87][88]. The detail of manually labeled dataset are given in **Table 4.4**.

Table 4.4: Manually Labeled Dataset

Dataset	Total Pairs	TYPE 1	TYPE 2	TYPE 3	TYPE 4
Krawitz	38	3	5	10	20
Roy et al.	120	4	17	47	52

After formulating all possible combinations, we train the classification model (random forest) with code segments labeled w, r, t original piece of code and predict the results on manually labeled code fragments.

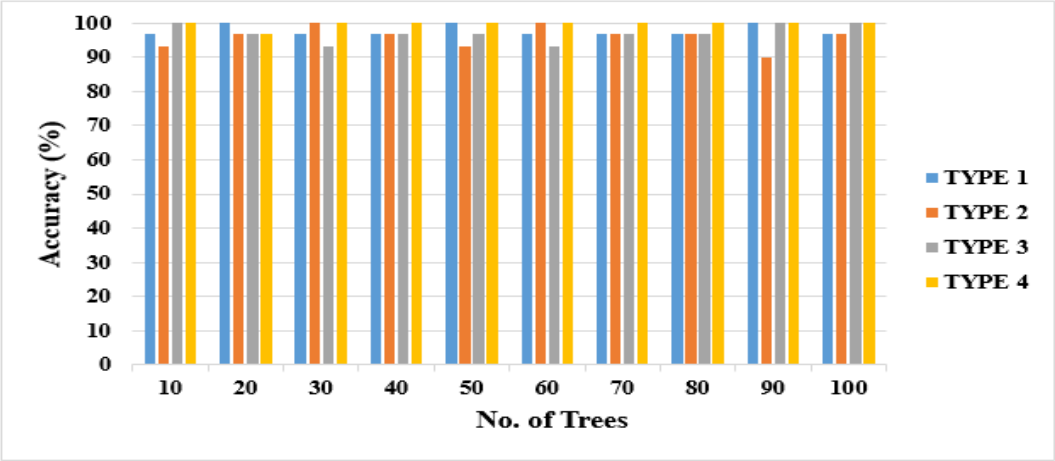


Figure 4.2: Detection rate on Krawtiz Dataset

We use random forest with varying no. of trees (10 … 100) to check the efficiency of our framework, as illustrated in **Figure 4.2** and **Figure 4.3**.

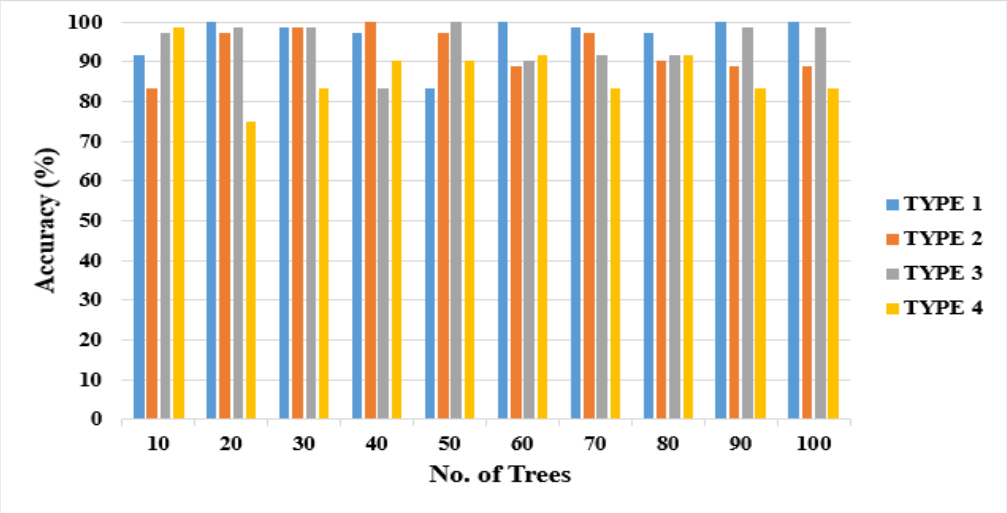


Figure 4.3: Detection rate on Roy et al. Dataset

Figure 4.2 and **Figure 4.3** shows that we get good results on all kinds of clones. For further validating the results of proposed framework, we perform 5-fold cross validation. The detail results are provided in **Figure 4.4** and **Figure 4.5**.

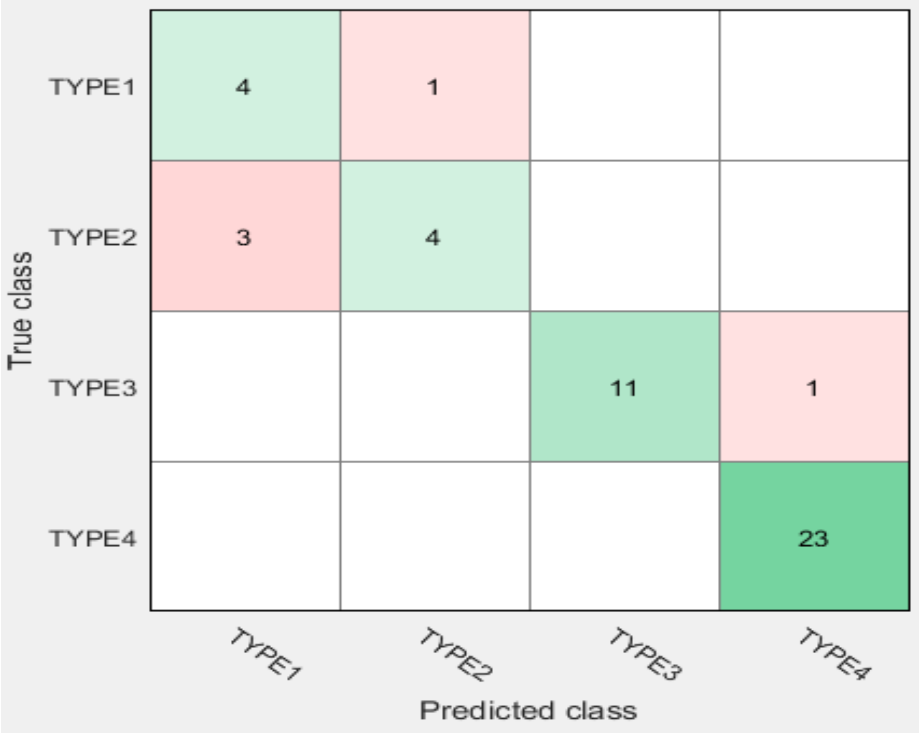


Figure 4.4: Effectiveness of the framework on Krawtiz dataset using 5 Fold Cross validation

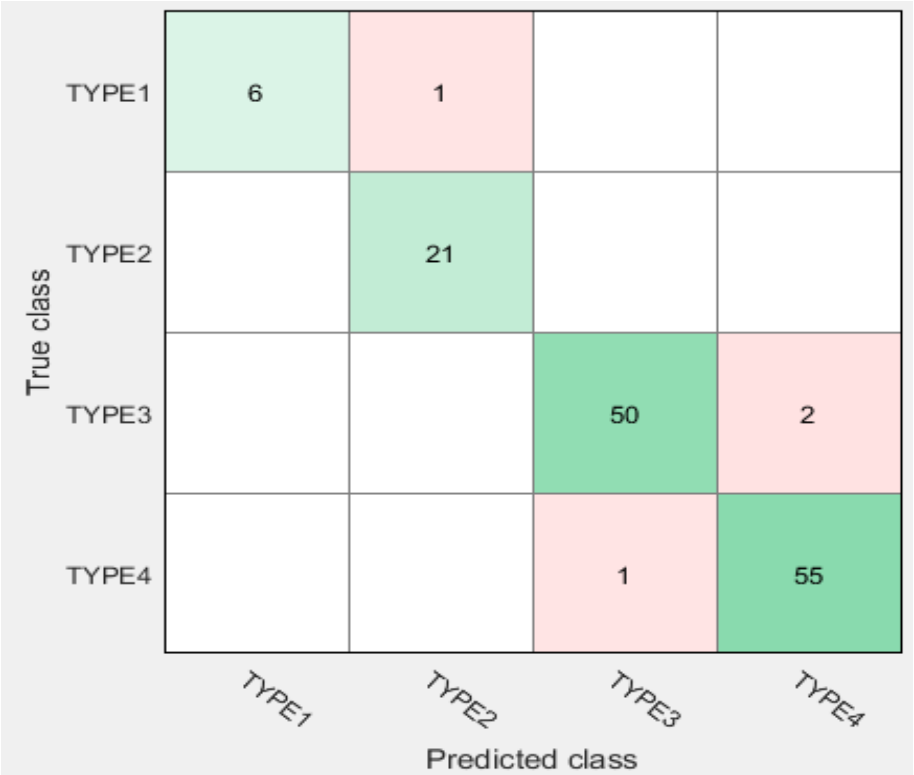


Figure 4.5: Effectiveness of the framework on Roy et al. dataset using K Fold Cross validation

Figure 4.4 and **Figure 4.5** shows that we get acceptable results on all types of clones by using 5-fold cross validation. These results are summarized in **Table 4.5**

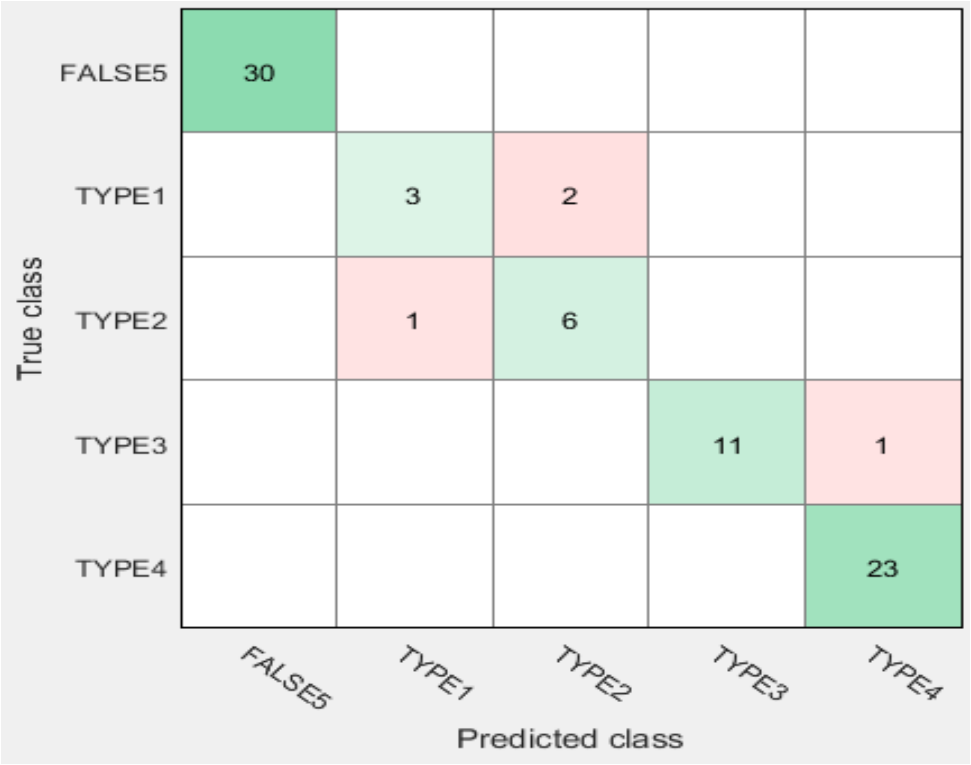
Table 4.5: Summery of extended experiment results

Name	TYPE 1	TYPE 2	TYPE 3	TYPE 4
Proposed framework	well	well	well	well

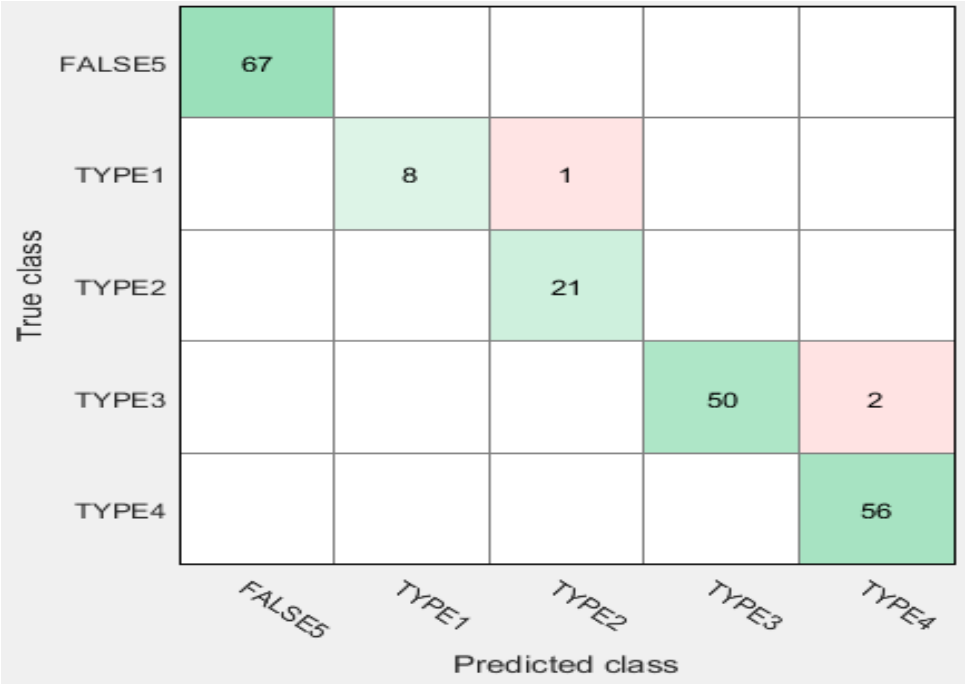
The datasets we utilized in this research work consist of only four types of clones (type 1, type 2, type 3 and type 4), they do not include false clones. To check the effectiveness of our framework, if a non-clone or false value occurs, we manually add some false clones in these datasets and make all possible combinations. The details of extended datasets with all possible combination is given in **Table 4.6**. The whole process is repeated to check these false clones and the detail results are explained in **Figure 4.6** given below.

Table 4.6: Extended dataset with all possible combinations

Dataset	Total Pairs	TYPE 1	TYPE 2	TYPE 3	TYPE 4	FALSE
Krawitz	77	5	7	12	23	30
Roy et al.	204	7	21	52	56	68



(a) Detection rate of False Clones in Krawitz dataset



(b) Detection rate of False Clones in Roy et al dataset

Figure 4.6: Effectiveness of framework on both datasets using 5-fold cross Validation

4.5. Classification Model

Classification model we utilized in our framework is random forest. Due to its simplicity and randomization process random forest has better performance than that of other classifiers e.g. support vector machine (SVM) and artificial neural network(ANN) [95]. Moreover, when comparison is made with other classification models random forest gives better accuracy [96]. The reason is that random forest is an ensambling approach. An ensambling approach gives better predictive performance as compared to single model [98]. For further verification of random forest give better results in terms of accuracy than other models, we apply other supervise learning models on our datasets and compare the predictive performance of these models with random forest in **Table 4.7**. Results indicate that random forest give better predictive performance than that of other models.

Table 4.7: Performance comparison of random forest with other models in term of accuracy

Sr.#	Classification Model	Dataset	Results
1	Random Forest	Krawitz	97.5%
		Roy et al.	92.6%
2	SVM	Krawitz	82.5%
		Roy et al.	92.6%
3	KNN	Krawitz	53.75%
		Roy et al.	60.94%
4	Decision Tree	Krawitz	50.58%
		Roy et al.	64.89%
5	Naive Bayes	Krawitz	77.78%
		Roy et al.	74.27%

4.5.1. Random Forest

Random forest is a supervised learning classification model, for the classification of dataset it utilizes decision trees. Growing an ensemble of trees and deciding the type of class by voting, significantly improves the classification accuracy. For growing these ensembles random vectors are constructed. Each tree is generated from one random vector. Random forest consists of classification trees. By analyzing output of these trees the classification problems are solved. The random forest prediction is determined by majority voting [20] as presented in **Figure 4. *7***.

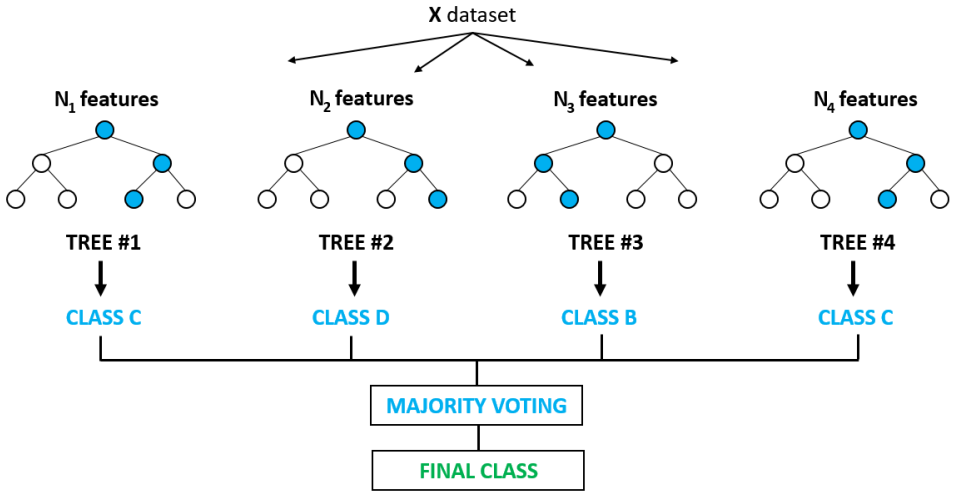


Figure 4. 7: Random Forest

4.6. Performance of Execution Time

The execution time of our framework is measured on both datasets. The execution time by varying the steps to detect code clones in the datasets is shown in **Figure 4.8**. The proposed framework can identify code clones in few seconds. Therefore, this framework can process millions line of code in feasible amount of time. This may indicate new clone detection approaches for large datasets.

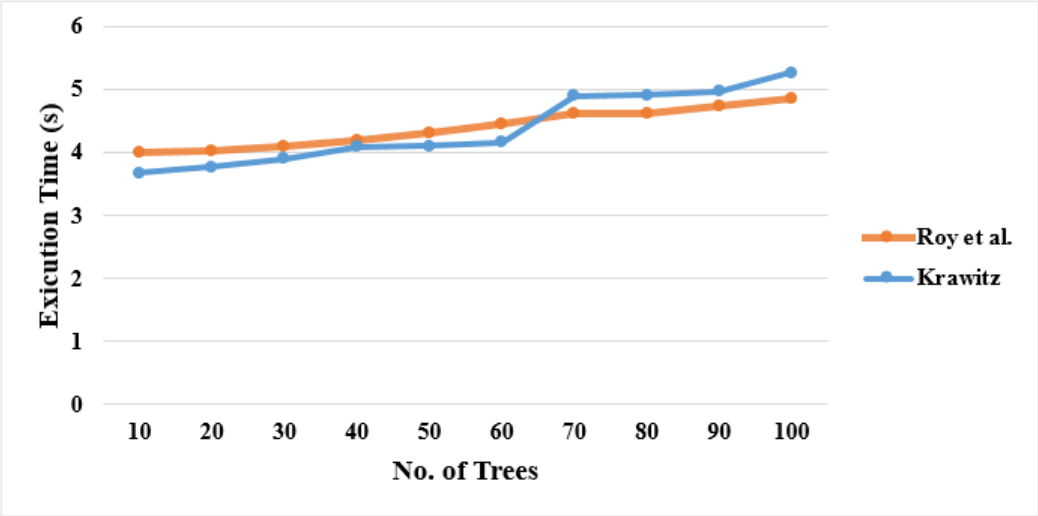


Figure 4.8: Execution time of proposed framework on both datasets

4.7. Experiments with different training and testing corpus

As an ideal approach, must be work on other datasets that are different from the dataset utilized during training stage. In order to figure out the competence of proposed approach when training and testing samples are from different datasets, we train the model on Krawitz dataset and test on Roy et al. and vice versa. The accuracy of our framework on different training and testing samples is shown in **Figure 4.9**.

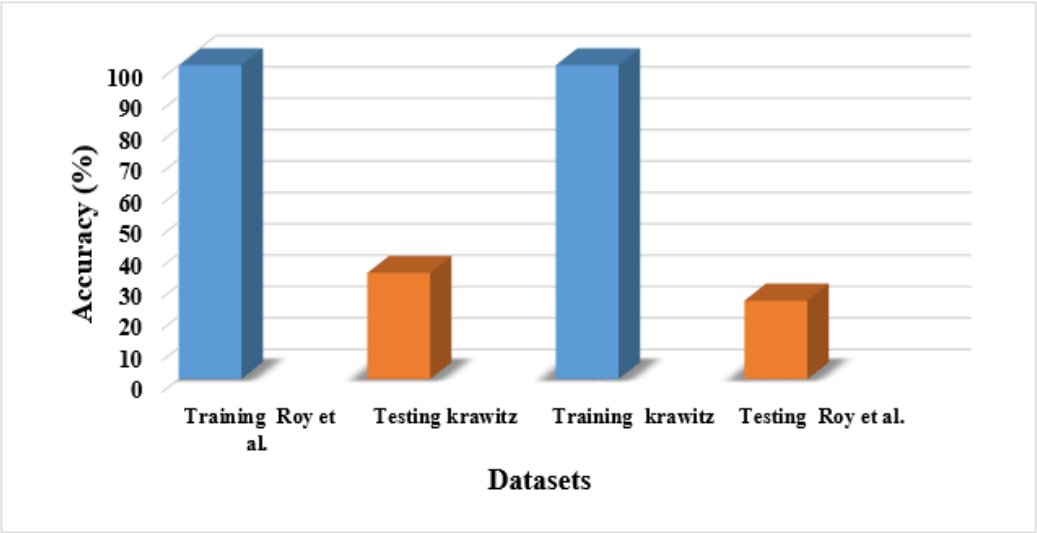


Figure 4.9: Effectiveness of framework for code clone detection when datasets are different training and testing.

Chapter 5

Discussion and Limitations

CHAPTER 5: DISCUSSION AND LIMITATIONS

This chapter consists of discussion and limitations of proposed framework. **Section 5.1** discuss the research work and **Section 5.2** consists of limitations of the proposed approach.

5.1. Discussion

This study, presents an approach to identify all (type1, type2, type3 and type4) types of clones (mainly type 4) in C programs. In this field few techniques and tools are developed that can detect these clones, but they utilize traditional methods which can detect type 4 clones with very low accuracy [9]. These clones are difficult to detect because these are semantically same but structurally different. From literature we find few (3 or 4) studies [48][67][71][82] that tried their best to detect all types of clones including type 4 with good results (accuracy, execution time) but they are applicable to java code only because parsers or compilers used by them are limited to java code.. However, current approaches are incompetent to find semantic clones along with other (type 1, type 2 and type 3) three types of clones with good results in programing languages (e.g. C/C++).

Our primary objective is to improve the accuracy of semantic or type 4 clones while not compromising the accuracies of other three types of clone in C program. For this purpose, two datasets Krawitz [84] and Roy et al. [9] are used. To measure the similarity between code blocks we parse each fragment of source code into AST. Different from manually extracting features for the detection of code clones, the proposed framework can automatically extract features by analyzing abstract syntax trees (ASTs) of source code. Moreover, supervised learning based classification model named random forest is used and conduct 2 set of experiments for code clone detection. Each set consists of pair instance feature using linear combination. The training and testing of classification model is performed by utilizing leave-one-out cross validation when dataset is partially labeled or consisting of small no. of combinations. After making all possible combinations the model is trained using dataset labeled with respect to original piece of code and tested on code fragments which are manually labeled for all possible clones. To further validate the accuracy, the model is trained and tested using K-fold cross validation where value of K varies from 10 to 2. Moreover, to check the effectiveness of framework if a non-clone occurs in the dataset we manually add some non-clones, and iterate the whole process. Our framework is generic in nature and can be extended to any other high programming language.

5.2. Limitations

The proposed approach also has some limitations which are explained in this section.

- The use of frama-c for compilation/ parsing (normalization, abstract syntax tree generation or feature extraction) is only applicable for C/C++ programs. For programming languages that utilize some kind of intermediate representation of source code like java and C#, these results might not be appropriate. However, we hope that by using compiler or parser of these languages our technique can work.
- All the C program files have to be parsed into ASTs and extract features from them. Therefore, all the C program files must have no error before parsing into AST and extracting features.

Chapter 6

Conclusion and Future Work

CHAPTER 6: CONCLUSION AND FUTURE WORK

This research work, presents an approach to find all types (type1, type2, type3 and type4) of clones in C program while our main focus is semantic or type 4 clones. For this purpose, we conduct an experiment by utilizing 2 datasets (Krawitz and Roy et al.). Different from manually extracting features for the detection of code clones, the proposed framework can automatically extract features by analyzing abstract syntax trees (ASTs) of source code. Afterwards, supervised learning based classification model has been used and conduct 2 sets of experiment for code clone detection. Each set consists of pair instance feature using linear combination. The training and testing of classification model is performed by utilizing leave-one-out cross validation when dataset is partially labeled and consisting of small no. of combinations. After making all possible combinations the model is trained using dataset labeled with respect to original piece of code and tested on dataset which is manually labeled for all possible clones. To further validate the accuracy, the model is trained and tested using K fold cross validation where value of K varies from 10 to 2. Furthermore, to check the effectiveness of framework if a non-clone occurs in the dataset we manually add some non-clones, and iterate the whole process.

The performance of our framework is compared with popular and state of the art code clone detectors that are used in several recent studies. Our results indicate that the proposed framework is comparable with other detectors in the detection of Type1 clones and superior in the detection of semantic or type 4 clones. However, proposed framework does not give acceptable results in finding Type2 and Type3 clones. Therefore, we perform some extended experiments and get valuable results on all types of clones.

This framework is limited to C/C++. However, we believe that it can be extended to any other high level programming language. Therefore, we plan to extend this approach to detect clones in other programming languages. The datasets utilized in this research work are small, so in future we conduct experiments on large datasets to check the scalability of this framework. Moreover, it is examined that source code of open source software systems is used for CCD. Similar to open source systems clones can also exist in commercial software systems. Therefore, in future commercial software systems should be target to check the validity of this approach on commercial level.

APPENDICES

Appendix A

Roy et al. Source Code

```
// Original Code - Cordy
void sumProdO(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i;
	for (i = 1; i <= n; i++)
	{
		sum = sum + i;
		prod = prod * i;
		foo2(sum, prod);
	}
}
void main()
{
	printf("\nsumProdO_Cordy: %lf ");
	sumProdO(4);
}

// Example 1A
void sumProd1A(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i;
	for (i = 1; i <= n; i++)
	{
		sum = sum + i;
		prod = prod * i;
		foo2(sum, prod);
	}
}
void main()
{
	printf("\nsumProd1A_Cordy: %lf ");
	sumProdO(4);
}


// Example 2A
void sumProd1B(int n) {
	double sum = 0.0; //C1
	double prod = 1.0; //C
	int i;
	for (i = 1; i <= n; i++)
	{
		sum = sum + i;
		prod = prod * i;
		foo2(sum, prod);
	}
}
void main()
{
```

```
	printf("\nsumProd1B_Cordy: %lf ");
	sumProdO(4);
}


// Example 2B
void sumProd1C(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i;
	for (i = 1; i <= n; i++)
	{
		sum = sum + i;
		prod = prod * i;
		foo2(sum, prod);
	}
}

void main()
{
	printf("\nsumProd1C_Cordy: %lf ");
	sumProdO(4);
}


// Example 2C
void sumProd2A(int n) {
	double s = 0; //C1
	double p = 1;
	int j;
	for (j = 1; j <= n; j++)
	{
		s = s + j;
		p = p * j;
		foo2(s, p);
	}
}
void main()
{
	printf("\nsumProd2A_Cordy: %lf ");
	sumProd2A(4);
}

Example 2B
void sumProd2B(int n) {
	double s = 0; //C1
	double p = 1;
	int j;
	for (j = 1; j <= n; j++)
	{
		s = s + j;
		p = p * j;
		foo2(p, s);
	}
}
{
	printf("\nsumProd2B_Cordy: %lf ");
	sumProd2B(4);
```

```
}

// Example 2C
void sumProd2C(int n) {
	int sum = 0; //C1
	int prod = 1;
	int i;
	for (i = 1; i <= n; i++)
	{
		sum = sum + i;
		prod = prod * i;
		foo2(sum, prod);
	}
}
{
	printf("\nsumProd2C_Cordy: %lf ");
	sumProd2C(4);
}

// Example 2D
void sumProd2D(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i;
	for (i = 1; i <= n; i++)
	{
		sum = sum + (i*i);
		prod = prod * (i*i);
		foo2(sum, prod);
	}
}
{
	printf("\nsumProd2D_Cordy: %lf ");
	sumProd2D(4);
}

// Example 3A
void sumProd3A(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i;
	for (i = 1; i <= n; i++)
	{
		sum = sum + i;
		prod = prod * i;
		foo3(sum, prod, n);
	}
}
{
	printf("\nsumProd3A_Cordy: %lf ");
	sumProd3A(4);
}

// Example 3B
void sumProd3B(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i;
```

```
	for (i = 1; i <= n; i++)
	{
		sum = sum + i;
		prod = prod * i;
		foo(prod);
	}
}
{
	printf("\nsumProd3B_Cordy: %lf ");
	sumProd3B(4);
}

// Example 3C -
void sumProd3C(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i;
	for (i = 1; i <= n; i++)
	{
		sum = sum + i;
		prod = prod * i;
		if ((n % 2) == 0) {
			foo2(sum, prod);
		}
	}
}

{
	printf("\nsumProd3C_Cordy: %lf ");
	sumProd3C(4);
}

// Example 3D
void sumProd3D(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i;
	for (i = 1; i <= n; i++)
	{
		sum = sum + i;
		//line deleted
		foo2(sum, prod);
	}
}
{
	printf("\nsumProd3D_Cordy: %lf ");
	sumProd3D(4);
}

// Example 3E
// For syntax purposes, the precise functionality was altered.
void sumProd3E(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i;
	for (i = 1; i <= n; i++)
	{
		if (i % 2 == 0)
```

```
		{
			sum += i;
		}
		prod = prod * i;
		foo2(sum, prod);
	}
}

{
	printf("\nsumProd3E_Cordy: %lf ");
	sumProd3E(4);
}

// Example 4a
void sumProd4A(int n) {
	double prod = 1.0;
	double sum = 1; //C1
	int i;
	for (i = 0; i <= n; i++)
	{
		sum = sum + i;
		prod = prod * i;

	}
}
void main()
{
	printf("\nsumProd4A_Cordy: %lf ");
	sumProd4A(4);
}

// Example 4B
void sumProd4A(int n) {
	double prod = 1.0;
	double sum = 0.0; //C1
	int i;
	for (i = 0; i <= n; i++)
	{
		sum += i;
		prod *= i;
		foo(sum, prod);

	}
}
void main()
{
	printf("\nsumProd4A_Cordy: %lf ");
	sumProd4B(4);
}

// Example 4C
void sumProd4C(int n) {
	double sum = 0.0; //C1
	double prod = 1.0;
	int i = 1;
	while (i <= n)
	{
		sum = sum + i;
```

```
                prod = prod * i;
                foo2(sum, prod);
        }
}
void main()
{
        printf("\nsumProd4C_Cordy: %lf ");
        sumProd4C(4);
}

// Example 4D
void sumProd4D(int n) {
        double sum = 0.0; //C1
        double prod = 1.0;
        int i = 0;
        while (i <= n)
        {
                sum = sum + i;
                prod = prod * i;
        }
}

void main()
{
        printf("\nsumProd4D_Cordy: %lf ");
        sumProd4D(4);
}
```

Krawitz Source Code

```
//Original
float Type1a_Krawitz(int n)
{
        int p = -1;
        int sum = 0;

        for (p = 0; p < n; p++)
        {
                sum += p;
        }

        if (n == 0) return sum;
        else return sum / n;
}

void main()
{
        printf("Type1a_Krawitz: %lf \n", Type1a_Krawitz(4));
}

//Type-1
float Type1a_Krawitz(int n)
{
        int p = -1;
        int sum = 0;
```

```
	for (p = 0; p < n; p++)
	{
		sum += p;
	}

	if (n == 0) return sum;
	else return sum / n;
}

void main()
{
	printf("Type1a_Krawitz: %lf \n", Type1a_Krawitz(4));
}

// Type 1
float Type1b_Krawitz(int n)
{
	int p = -1;
	int sum = 0;

	//this is a comment that is not in any other method()
	for (p = 0; p < n; p++)
		sum += p;

	if (n == 0)
		return sum;
	else
		return sum / n;
}



void main()
{
	printf("Type1b_Krawitz: %lf \n", Type1b_Krawitz(4));
}

//Type-2
float Type2a_Krawitz(int n)
{
	int q = -1;
	double sum = 0;

	for (q = 0; q < n; q++)
	{
		sum += q;
	}

	if (n == 0) return sum;
	else return sum / n;
}

void main()
{
	printf("Type2a_Krawitz: %lf \n", Type2a_Krawitz(4));
}
//Type-2
float Type2b_Krawitz(int t)
```

```
{
	int p = -1;
	int tot = 0;

	//this is a comment that is not the same as any other comment
	for (p = 0; p < t; p++)
		tot += p;

	if (t == 0)
		return tot;
	else
		return tot / t;
}

void main()
{
	printf("Type2b_Krawitz: %lf \n", Type2b_Krawitz(4));
}
// type 3
float Type3a_Krawitz(int n)
{
	int q = -1;
	double sum = 0;

	q = 0;
	while (q < n)
	{
		sum += q;
		q++;
	}

	if (n == 0) return sum;
	else return sum / n;
}


void main()
{
	printf("Type3a_Krawitz: %lf \n", Type3a_Krawitz(4));
}

// type 3
float Type3b_Krawitz(int t)
{
	int p = -1, tot = 0;

	//this is another unique comment
	for (p = 0; p < t; p++)
		tot += p;

	if (t == 0)
		return (double)tot;
	else
		return (double)tot / t;
}


void main()
```

```
{
	printf("Type3b_Krawitz: %lf \n", Type3b_Krawitz(4));
}
//Type-4
double Type4a_Krawitz(int limit) {
	double* d;
	double tot = 0;
	int n;
	//to prevent stack overflow when large random values are input
	if (limit > 1000 || limit < 1)
		limit = 1;

	d = (double*)malloc(limit * sizeof(double));

	for (n = 0; n < limit; n++)
		d[n] = n * n * n;

	for (n = 0; n < limit; n++)
		tot += d[n];
	free((void*)d);
	return tot; 1
}



void main()
{
	printf("Type4a_Krawitz: %lf \n", Type4a_Krawitz(4));
}
// Type 4
double Type4b2_Krawitz(char s, int limit, double tot, int n) {

	//to prevent stack overflow when large random values are input
	if (limit > 1000 || limit < 1)
		limit = 1000;

	if (n < limit)
		tot = Type4b2_Krawitz('-', limit, tot + n * n*n, ++n);

	return tot;}

void main()
{
	printf("Type4b_Krawitz: %lf \n", Type4b_Krawitz(4));
}
// Type 4
double Type4b_Krawitz(int limit) {

	//to prevent stack overflow when large random values are input
	if (limit > 1000 || limit < 1)
		limit = 1000;

	return Type4b2_Krawitz('-', limit, 0, 0);
}

void main()
{
	printf("Type4b2_Krawitz: %lf \n", Type4b2_Krawitz('-', 3, 3.0, 4));}
```

REFERENCES

[1] Baker, Brenda S. "On finding duplication and near-duplication in large software systems." In *Reverse Engineering, 1995. Proceedings of 2nd Working Conference on*, pp. 86-95. IEEE, 1995.

[2] Ducasse, Stéphane, Matthias Rieger, and Serge Demeyer. "A language independent approach for detecting duplicated code." In *Software Maintenance, 1999. (ICSM'99) Proceedings. IEEE International Conference on*, pp. 109-118. IEEE, 1999.

[3] Krinke, Jens. "Identifying similar code with program dependence graphs." In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pp. 301-309. IEEE, 2001.

[4] Baxter, Ira D., Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. "Clone detection using abstract syntax trees." In *Software Maintenance, 1998. Proceedings. International Conference on*, pp. 368-377. IEEE, 1998.

[5] Ducasse, Stéphane, Matthias Rieger, and Serge Demeyer. "A language independent approach for detecting duplicated code." In *Software Maintenance, 1999. (ICSM'99) Proceedings. IEEE International Conference on*, pp. 109-118. IEEE, 1999.

[6] Komondoor, Raghavan, and Susan Horwitz. "Using slicing to identify duplication in source code." In *International static analysis symposium*, pp. 40-56. Springer, Berlin, Heidelberg, 2001.

[7] Roy, Chanchal Kumar, and James R. Cordy. "A survey on software clone detection research." *Queen's School of Computing TR* 541, no. 115 (2007): 64-68.

[8] Saini, Neha, and Sukhdip Singh. "Code Clones: Detection and Management." *Procedia Computer Science* 132 (2018): 718-727.

[9] Roy, Chanchal K., James R. Cordy, and Rainer Koschke. "Comparison and evaluation of techniques and tools: A qualitative approach." *Science of computer programming* 74, no. 7 (2009): 470-495.

[10] Roy, Chanchal K., and James R. Cordy. "An empirical study of function clones in open source software." In *2008 15th Working Conference on Reverse Engineering*, pp. 81-90. IEEE, 2008.

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

[12] Kamiya, Toshihiro. "The official CCFinderX website." *URL http://www. ccfinder. net/ccfinderx. html Last accessed November* (2008).URL http://www.ccfinder.net/ccfinderx.html.

[13] Higo, Yoshiki, Yasushi Ueda, Toshihro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "On software maintenance process improvement based on code clone analysis." In *International Conference on Product Focused Software Process Improvement*, pp. 185-197. Springer, Berlin, Heidelberg, 2002.

[14] Tool Simian. Last Accessed November 2008. URL http://www.redhillconsulting.com.au/products/simian/.

[15] Li, Zhenmin, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. "CP-Miner: Finding copy-paste and related bugs in large-scale software code." *IEEE Transactions on software Engineering*32, no. 3 (2006): 176-192.

[16] Johnson, J. Howard. "Visualizing textual redundancy in legacy source." In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, p. 32. IBM Press, 1994.

[17] Johnson, J. Howard. "Substring Matching for Clone Detection and Change Tracking." In *ICSM*, vol. 94, pp. 120-126. 1994.

[18] Davey, Neil, Paul Barson, Simon Field, Ray Frank, and D. Tansley. "The development of a software clone detector." *International Journal of Applied Software Technology* (1995).

[19] Di Lucca, Giuseppe A., Massimiliano Di Penta, and Anna Rita Fasolino. "An approach to identify duplicated web pages." In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pp. 481-486. IEEE, 2002.

[20] Breiman, Leo. "Random Forests, Vol. 45." *Mach Learn* 1 (2001).

[21] Ragkhitwetsagul, Chaiyong, and Jens Krinke. "Using compilation/decompilation to enhance clone detection." In *11th International Workshop on Software Clone (IWSC'17)*, vol. 11, pp. 8-14. IEEE, 2017.

[22] Kim, Seulbae, and Heejo Lee. "Software systems at risk: An empirical study of cloned vulnerabilities in practice." *Computers & Security* (2018).

[23] Jadon, Shruti. "Code clones detection using machine learning technique: Support vector machine." In *Computing, Communication and Automation (ICCCA), 2016 International Conference on*, pp. 399-303. IEEE, 2016.

[24] D Yu, Dongjin, Jie Wang, Qing Wu, Jiazha Yang, Jiaojiao Wang, Wei Yang, and Wei Yan. "Detecting Java Code Clones with Multi-granularities Based on Bytecode." In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, pp. 317-326. IEEE, 2017.

[25] Kim, Seulbae, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. "VUDDY: a scalable approach for vulnerable code clone discovery." In *Security and Privacy (SP), 2017 IEEE Symposium on*, pp. 595-614. IEEE, 2017.

[26] Y Nakamura, Yuta, Eunjong Choi, Norihiro Yoshida, Shusuke Haruna, and Katsuro Inoue. "Towards detection and analysis of interlanguage clones for multilingual web applications." In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 3, pp. 17-18. IEEE, 2016.

[27] Lyu, Fang, Yapin Lin, Junfeng Yang, and Junhai Zhou. "SUIDroid: An Efficient Hardening-Resilient Approach to Android App Clone Detection." In *Trustcom/BigDataSE/I SPA, 2016 IEEE*, pp. 511-518. IEEE, 2016.

[28] Xue, Hongfa, Guru Venkataramani, and Tian Lan. "Clone-hunter: accelerated bound checks elimination via binary code clone detection." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 11-19. ACM, 2018.

[29] Chen, Jian, Manar H. Alalfi, Thomas R. Dean, and Ying Zou. "Detecting android malware using clone detection." *Journal of Computer Science and Technology* 30, no. 5 (2015): 942-956.

[30] Thaller, Hannes, Rudolf Ramler, Josef Pichler, and Alexander Egyed. "Exploring code clones in programmable logic controller software." *arXiv preprint arXiv:1706.03934* (2017).

[31] Newman, Christian D., Tessandra Sage, Michael L. Collard, Hakam W. Alomari, and Jonathan I. Maletic. "srcSlice: a tool for efficient static forward slicing." In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pp. 621-624. IEEE, 2016.

[32] Liu, Zhen, Qiang Wei, and Yan Cao. "VFDETECT: A vulnerable code clone detection system based on vulnerability fingerprint." In *Information Technology and Mechatronics Engineering Conference (ITOEC), 2017 IEEE 3rd*, pp. 548-553. IEEE, 2017.

[33] Reddivari, Sandeep, and Mohammed Salman Khan. "CloneTM: A Code Clone Detection Tool Based on Latent Dirichlet Allocation." In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 930-931. IEEE, 2019.

[34] Ghosh, Akash, and Sandeep Kaur Kuttal. "Semantic Clone Detection: Can Source Code Comments Help?." In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 315-317. IEEE, 2018.

[35] Nishi, Manziba Akanda, and Kostadin Damevski. "Scalable code clone detection and search based on adaptive prefix filtering." *Journal of Systems and Software* 137 (2018): 130-142.

[36] Tekchandani, Rajkumar, Rajesh Bhatia, and Maninder Singh. "Code clone genealogy detection on e-health system using Hadoop." *Computers & Electrical Engineering* 61 (2017): 15-30.

[37] Farhadi, Mohammad Reza, Benjamin CM Fung, Yin Bun Fung, Philippe Charland, Stere Preda, and Mourad Debbabi. "Scalable code clone search for malware analysis." *Digital Investigation* 15 (2015): 46-60.

[38] Wang, Pengcheng, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. "CCAligner: a token based large-gap clone detector." In *Proceedings of the 40th International Conference on Software Engineering*, pp. 1066-1077. ACM, 2018.

[39] Yuki, Yusuke, Yoshiki Higo, and Shinji Kusumoto. "A technique to detect multi-grained code clones." In *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*, pp. 1-7. IEEE, 2017.

[40] Sajnani, Hitesh, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. "SourcererCC: scaling code clone detection to big-code." In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 1157-1168. IEEE, 2016.

[41] Semura, Yuichi, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. "CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization." In *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*, pp. 654-659. IEEE, 2017.

[42] Li, Liuqing, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. "CCLearner: A Deep Learning-Based Clone Detection Approach." In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pp. 249-260. IEEE, 2017.

[43] Semura, Yuichi, Norihiro Yoshiday, Eunjong Choi, and Katsuro Inoue. "Multilingual Detection of Code Clones Using ANTLR Grammar Definitions." In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 673-677. IEEE, 2018.

[44] Uemura, Kyohei, Akira Mori, Eunjong Choi, and Hajimu Iida. "Tracking Method-Level Clones and a Case Study." In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pp. 27-33. IEEE, 2019.

[45] Yang, Yanming, Zhilei Ren, Xin Chen, and He Jiang. "Structural Function Based Code Clone Detection Using a New Hybrid Technique." In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pp. 286-291. IEEE, 2018.

[46] Pati, Jayadeep, Babloo Kumar, Devesh Manjhi, and Kaushal K. Shukla. "A Comparison Among ARIMA, BP-NN, and MOGA-NN for Software Clone Evolution Prediction." *IEEE Access* 5 (2017): 11841-11851

[47] Chodarev, Sergej, Emília Pietriková, and Ján Kollár. "Haskell clone detection using pattern comparing algorithm." In *Engineering of Modern Electric Systems (EMES), 2015 13th International Conference on*, pp. 1-4. IEEE, 2015.

[48] Zeng, Jie, Kerong Ben, Xiaowei Li, and Xian Zhang. "Fast Code Clone Detection Based on Weighted Recursive Autoencoders." *IEEE Access* 7 (2019): 125062-125078.

[49] Tsunoda, Masateru, Yasutaka Kamei, and Atsushi Sawada. "Assessing the Differences of Clone Detection Methods Used in the Fault-prone Module Prediction." In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 3, pp. 15-16. IEEE, 2016.

[50] Svajlenko, Jeffrey, and Chanchal K. Roy. "Fast and flexible large-scale clone detection with CloneWorks." In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pp. 27-30. IEEE, 2017.

[51] Sudhamani, M., and Lalitha Rangarajan. "Code clone detection based on order and content of control statements." In *Contemporary Computing and Informatics (IC3I), 2016 2nd International Conference on*, pp. 59-64. IEEE, 2016.

[52] Svajlenko, Jeffrey, and Chanchal K. Roy. "Cloneworks: A fast and flexible large-scale near-miss clone detection tool." In *Proceedings of the 39th International Conference on Software Engineering Companion*, pp. 177-179. IEEE Press, 2017.

[53] Haque, Syed Mohd Fazalul, V. Srikanth, and E. Sreenivasa Reddy. "Generic code Cloning method for detection of Clone code in software Development." In *Data Mining and Advanced Computing (SAPIENCE), International Conference on*, pp. 335-339. IEEE, 2016.

[54] Ragkhitwetsagul, Chaiyong, Jens Krinke, and Bruno Marnette. "A picture is worth a thousand words: Code clone detection based on image similarity." In *Software Clones (IWSC), 2018 IEEE 12th International Workshop on*, pp. 44-50. IEEE, 2018.

[55] Sudhamani, M., and Lalitha Rangarajan. "Structural similarity detection using structure of control statements." *Procedia Computer Science* 46 (2015): 892-899.

[56] Yu, Dongjin, Jiazha Yang, Xin Chen, and Jie Chen. "Detecting Java Code Clones Based on Bytecode Sequence Alignment." *IEEE Access* 7 (2019): 22421-22433.

[57] Sudhamani, M., and Lalitha Rangarajan. "Code similarity detection through control statement and program features." *EXPERT SYSTEMS WITH APPLICATIONS* 132 (2019): 63-75.

[58] Wang, Min, Pengcheng Wang, and Yun Xu. "CCSharp: An Efficient Three-Phase Code Clone Detector Using Modified PDGs." In *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*, pp. 100-109. IEEE, 2017.

[59] Sabi, Yusuke, Yoshiki Higo, and Shinji Kusumoto. "Rearranging the order of program statements for code clone detection." In *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*, pp. 1-7. IEEE, 2017.

[60] Crussell, Jonathan, Clint Gibler, and Hao Chen. "Andarwin: Scalable detection of android application clones based on semantics." *IEEE Transactions on Mobile Computing* 14, no. 10 (2015): 2007-2019.

[61] Sargsyan, Sevak, Sh Kurmangaleev, A. Belevantsev, and Arutyun Avetisyan. "Scalable and accurate detection of code clones." *Programming and Computer Software* 42, no. 1 (2016): 27-33.

[62] Hu, Yikun, Yuanyuan Zhang, Juanru Li, and Dawu Gu. "Binary code clone detection across architectures and compiling configurations." In *Proceedings of the 25th International Conference on Program Comprehension*, pp. 88-98. IEEE Press, 2017.

[63] Kamalpriya, C. M., and Paramvir Singh. "Enhancing program dependency graph based clone detection using approximate subgraph matching." In *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*, pp. 1-7. IEEE, 2017.

[64] Avetisyan, Arutyun, Shamil Kurmangaleev, Sevak Sargsyan, Mariam Arutunian, and Andrey Belevantsev. "LLVM-based code clone detection framework." In *Computer Science and Information Technologies (CSIT), 2015*, pp. 100-104.

[65] Singh, Gurpreet. "To enhance the code clone detection algorithm by using hybrid approach for detection of code clones." In *Intelligent Computing and Control Systems (ICICCS), 2017 International Conference on*, pp. 192-198.

[66] Misu, Md Rakib Hossain, and Kazi Sakib. "Interface Driven Code Clone Detection." In *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*, pp. 747-748. IEEE, 2017.

[67] Sheneamer, Abdullah, and Jugal Kalita. "Semantic clone detection using machine learning." In *Machine Learning and Applications (ICMLA), 2016 15th IEEE International Conference on*, pp. 1024-1028. IEEE, 2016.

[68] Vislavski, Tijana, Gordana Rakic, Nicolás Cardozo, and Zoran Budimac. "LICCA: A tool for cross-language clone detection." In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 512-516. IEEE, 2018.

[69] Misu, Md Rakib Hossain, Abdus Satter, and Kazi Sakib. "An Exploratory Study on Interface Similarities in Code Clones." In *Software Engineering Conference Workshops (APSECW), 2017 24th Asia-Pacific*, pp. 126-133. IEEE, 2017.

[70] Akram, Junaid, Zhendong Shi, Majid Mumtaz, and Ping Luo. "DroidCC: A Scalable Clone Detection Approach for Android Applications to Detect Similarity at Source Code Level." In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pp. 100-105. IEEE, 2018.

[71] Matsushita, Tsubasa, and Isao Sasano. "Detecting code clones with gaps by function applications." In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pp. 12-22. ACM, 2017.

[72] Kodhai, Egambaram, and Selvadurai Kanmani. "Method-level code clone detection through LWH (Light Weight Hybrid) approach." *Journal of Software Engineering Research and Development* 2, no. 1 (2014): 12.

[73] Tekchandani, Rajkumar, Rajesh Bhatia, and Maninder Singh. "Semantic code clone detection for Internet of things applications using reaching definition and liveness analysis." *The Journal of Supercomputing* 74, no. 9 (2018): 4199-4226.

[74] Uemura, Kyohei, Akira Mori, Kenji Fujiwara, Eunjong Choi, and Hajimu Iida. "Detecting and analyzing code clones in HDL." In *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*, pp. 1-7. IEEE, 2017

[75] Nasirloo, Hamid, and Fatemeh Azimzadeh. "Semantic code clone detection using abstract memory states and program dependency graphs." In *2018 4th International Conference on Web Research (ICWR)*, pp. 19-27. IEEE, 2018.

[76] Singh, Manu, and Vidushi Sharma. "Detection of file level clone for high level cloning." *Procedia Computer Science* 57 (2015): 915-922.

[77] Sheneamer, Abdullah, and Jugal Kalita. "Code clone detection using coarse and fine-grained hybrid approaches." In *Intelligent Computing and Information Systems (ICICIS), 2015 IEEE Seventh International Conference on*, pp. 472-480. IEEE, 2015.

[78] White, Martin, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. "Deep learning code fragments for code clone detection." In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 87-98. ACM, 2016.

[79] Ragkhitwetsagul, Chaiyong, Jens Krinke, and David Clark. "A comparison of code similarity analysers." *Empirical Software Engineering* 23, no. 4 (2018): 2464-2519.

[80] Ghofrani, Javad, Mahdi Mohseni, and Arezoo Bozorgmehr. "A conceptual framework for clone detection using machine learning." In *Knowledge-Based Engineering and Innovation (KBEI), 2017 IEEE 4th International Conference on*, pp. 0810-0817. IEEE, 2017.

[81] Liu, Jinze, Tao Wang, Chenhui Feng, Huaimin Wang, and Dongsheng Li. "A Large-Gap Clone Detection Approach Using Sequence Alignment via Dynamic Parameter Optimization." *IEEE Access* 7 (2019): 131270-131281.

[82] Yu, Hao, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. "Neural detection of semantic code clones via tree-based convolution." In *Proceedings of the 27th International Conference on Program Comprehension*, pp. 70-80. IEEE Press, 2019.

[83] Baxter, Ira D., Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. "Clone detection using abstract syntax trees." In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368-377. IEEE, 1998.

[84] Krawitz, Ronald M. *Code clone discovery based on functional behavior*. Nova Southeastern University, 2012.

[85] Gabel, Mark, Lingxiao Jiang, and Zhendong Su. "Scalable detection of semantic clones." In *Proceedings of the 30th international conference on Software engineering*, pp. 321-330. ACM, 2008.

[86] Kamiya, Toshihiro, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code." *IEEE Transactions on Software Engineering* 28, no. 7 (2002): 654-670.

[87] Komondoor, Raghavan, and Susan Horwitz. "Using slicing to identify duplication in source code." In *International static analysis symposium*, pp. 40-56. Springer, Berlin, Heidelberg, 2001.

[88] Li, Zhenmin, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. "CP-Miner: Finding copy-paste and related bugs in large-scale software code." *IEEE Transactions on software Engineering* 32, no. 3 (2006): 176-192.

[89] Bellon, Stefan, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. "Comparison and evaluation of clone detection tools." *IEEE Transactions on software engineering* 33, no. 9 (2007): 577-591.

[90] Kontogiannis, Kostas. "Evaluation experiments on the detection of programming patterns using software metrics." In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pp. 44-54. IEEE, 1997.

[91] Kapser, Cory, and Michael W. Godfrey. "Aiding comprehension of cloning through categorization." In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, pp. 85-94. IEEE, 2004.

[92] Mayrand, Jean, Claude Leblanc, and Ettore Merlo. "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics." In *icsm*, vol. 96, p. 244. 1996.

[93] Aversano, Lerina, Luigi Cerulo, and Massimiliano Di Penta. "How clones are maintained: An empirical study." In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pp. 81-90. IEEE, 2007.

[94] C.K. Roy, J.R. Cordy, WCRE'08 Clones. Last Accessed November 2008. http://www.cs.queensu.ca/home/stl/download/NICADOutput/.

[95] Dogru, Nejdet, and Abdulhamit Subasi. "Traffic accident detection using random forest classifier." In *2018 15th Learning and Technology Conference (L&T)*, pp. 40-45. IEEE, 2018.

[96] Kumar, M. Suresh, V. Soundarya, S. Kavitha, E. S. Keerthika, and E. Aswini. "Credit Card Fraud Detection Using Random Forest Algorithm." In *2019 3rd International Conference on Computing and Communications Technologies (ICCCT)*, pp. 149-153. IEEE, 2019.

[97] Kodhai, Egambaram, and Selvadurai Kanmani. "Method-level code clone detection through LWH (Light Weight Hybrid) approach." *Journal of Software Engineering Research and Development* 2, no. 1 (2014): 12.

[98] Abuassba, Adnan OM, Dezheng Zhang, Xiong Luo, Ahmad Shaheryar, and Hazrat Ali. "Improving classification performance through an advanced ensemble based heterogeneous extreme learning machines." *Computational intelligence and neuroscience* 2017 (2017).