Enhancing Match Detection Process Using Chi-Square Equation for Improving Type-3 and Type-4 Clones in Java Applications

Noormaizzattul Akmaliza Abdullah[1], Al-Fahim Mubarak-Ali[2], Mohd Azwan Mohamad Hamza[3], Siti Salwani Yaacob[4]
Faculty of Computing, Universiti Malaysia Pahang Al-Sultan Abdullah, Pekan, Pahang, Malaysia[1, 3, 4]
Centre For Artificial Intelligence & Data Science, Universiti Malaysia Pahang Al-Sultan Abdullah, Lebuh Persiaran Tun Khalil, Yaakob, 26300 Gambang, Kuantan, Pahang[2]

***Abstract*—Generic Code Clone Detection (GCCD) is a code clone detection model that use distance measure equation, enabling detection of all types of code clones, naming clone Type-1, Type-2, Type-3 and Type-4 in Java programming language applications. However, the detection process of GCCD did not focus on detecting clones of Type-3 and Type-4. Hence, this paper suggested two experiments to incorporate enhancements to the GCCD in order to improve the detection rate of clone Type-3 and clone Type-4. The implementation of Chi-square distance in the match detection process produced a significant result increase in the experiment specifically on clones Type-3 and Type-4, in comparison with the Euclidean distance in GCCD, which allows the increase of detection rate due to the dissimilarity of the distance measures. Based on the results, the suggested enhancement using Chi-square distance on match detection process outperforms GCCD in terms of improving code clone detection results based on clone Type-3 and Type-4, as the objectives for each experiment are carried, contributes to the research on improving the code clone detection result.**

***Keywords*—Code clone detection; distance measure; Java language; Chi-square; computational intelligence**

I. Introduction

The practice of copying code is known as code cloning and the clone being duplicated is a code clone [1]–[3]. 60% developers went on searching code examples every day as to reduce the development time process [4] and which leads to code cloning. However, the integrity of certain developer cannot be underestimated as code examples could be implemented to the system instead of coding a new code fragment which lead to code cloning. Cost and programmer's limitation, templating and many more could also be the reasons for code cloning [4]–[6]. These reasons for code cloning could lead to drawbacks on software development and maintenance. The increase of maintenance cost, bug propagation, computational complexity and vulnerability proneness [5]–[8]. The inadequacy of programming language could also affect the code cloning [9], [10]. Java is a free programming language that developed open-source software applications. A study mentioned that 6% of 512000 lines of codes in Java applications are code clones [11]. The study concluded that the reason of code clones was the stake-holder's demand and deficiency in Java generic modules.

Code clones are generally categorized into four distinct types [12]–[14]; Type-1, Type-2, Type-3 and Type-4 (Fig. 1). Clone Type-1 is known for exact matches. These are code fragments that are identical, except for differences in whitespace and comments. Type-1 clones are the simplest to detect since they involve straightforward duplication without any modifications to the actual code logic. Clone Type-2 is renamed clones. In these clones, the code fragments are identical except for variations in identifiers, literals, types, or other superficial changes. While the overall structure and logic remain the same, these changes can make detection more complex than Type-1 clones. Clone Type-3 is modified clones. These are more complex clones where the duplicated code has undergone modifications such as adding or removing lines of code, altering control structures, or making other significant changes. Despite these modifications, the underlying logic or functionality of the code remains similar, making Type-3 clones challenging to detect. Finally, clone Type-4, the semantic clones. The most difficult to detect, Type-4 clones involve code fragments that perform the same functionality but are implemented using entirely different syntax or structures. These clones require deep semantic analysis to identify, as they do not share visible structural similarities with the original code.

Several major code clone approaches are prior to undertaking. The six major code clone approaches include text-based approaches, token-based approaches, metric-based approaches, tree-based approaches, graph-based approaches, and hybrid approaches. However, most approaches are incapable of recognizing all code clones [15]. For instance, token-based approach code detection tool such as NiCad [16],[17] detect the lexical part of the source code without considering the semantic information, which then prompted a poor detection result of clone Type-3 [18]. As a response, a code clone detection model was built for detecting code clones effectively. A model for the detection of the code clone incorporates structural process that combine several approaches or tools for the detection of clones. Several models that have existing in the code clone domain are the Generic Clone Model [19], the Generic Pipeline Model [20], Unified Clone Model [21], as well as the Generic Code Clone Detection (GCCD) Model [11]. Multiple researchers studied these four models in order to develop an effective code clone detection model. For instance, an enhancement was made in a

study on Generic Pipeline Model whereby they concatenated the source code file through Divide and Conquer method to enhance the load processing speed, by dividing the file into sub files [22]. They managed to decrease the model's runtime performance and increase the model's performance fully. Another study is on the GCCD model where they enhanced the pre-processing and parameterization process of GCCD [9], [23]. They managed to reduce the pre-processing rules and finding the best weightage to produce a better code clone detection model.

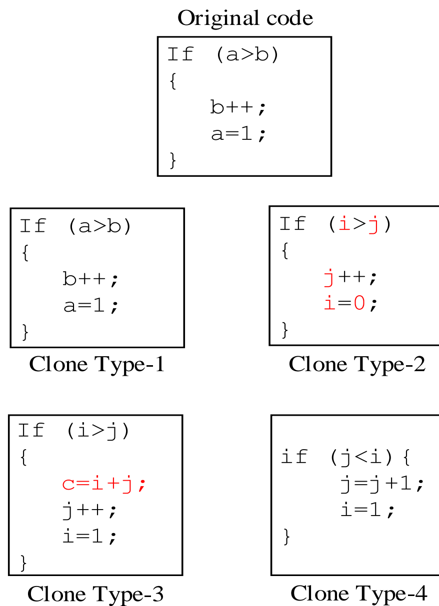

Fig. 1. Example of code clone Type-1, Type-2, Type-3 and Type-4.

The preliminary goal of this work is for enhancing the GCCD in order to produce a reliable code clone detection result, specifically Type-3 and Type-4. Therefore, this work will focus on;

- The experiment is an enhancement on match detection process using different distance measure equation such as Manhattan distance, Squared Euclidean distance, Half-squared Euclidean distance, Chi-square distance, in order to find the highest detection result of Type-3 and Type-4.
- The comparative analysis of the result between GCCD as an existing model and the proposed enhancement towards GCCD in terms of Type-3 and Type-4.

II. Related Works

Generic Code Clone Detection (GCCD) is a tool commenced for an aim to detect code clones in Java language application [11]. This model was developed targeting to detect code clone Type-1, Type2, Type-3 and Type-4. The purpose of GCCD is to provide a generality approach for identifying all sorts of code clone. GCCD is constructed with a structure of five processes (Fig. 2). The processes are pre-processing process, transformation process, parameterization process, categorization process and match detection process.

A. Pre-Processing Process

This process standardizes the source code as input for the detection process. There are five pre-processing rules applied to the source code in order to remove unnecessary elements that may conflict with the code clone detection result. The first pre-processing rule removes packages and import statements from the source code. Then, the second pre-processing rule is followed by removing comment lines as the comments are considered as instruction or guidance to the programmer only, which is not conflicting with the source code. The third pre-processing rule removes empty lines, which normally do not hold any source code and act as a method to visualize a clean look in coding. After that, the code is then regularized in the fourth pre-processing rule by replacing all function access modifier to public access modifier. This part of the rule act as a constant value for producing metrics in parameterization process. The fifth rule is to convert all uppercase letters in the original source code to lowercase letters for reducing the difference for detecting code clones. The essence of the source code has been filtered and the source code has become source unit.

B. Transformation Process

This process converts the source units into measurable units by substituting the source units with numerical value. The source unit is substituted one by one based on the position of alphabets. For instance, the alphabet *p* is substituted to 16 as the numerical value. Taking an example of word of the source unit from the Fig. 3, we can see that 'public' is substituted with value 162102120903.

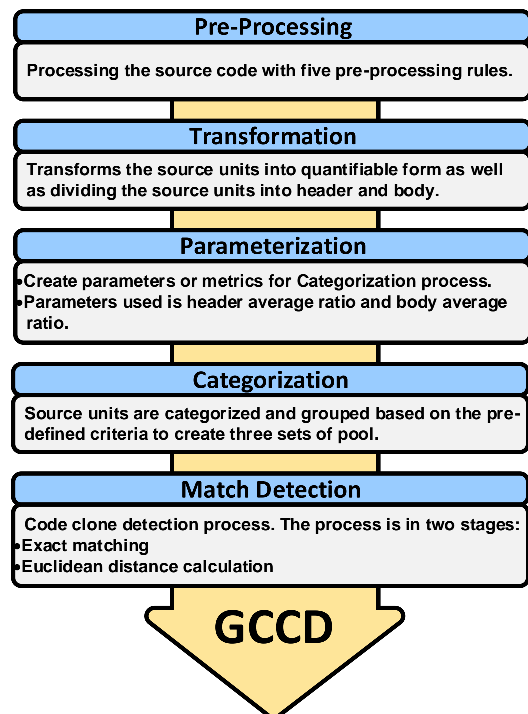

Fig. 2. Structure process of GCCD.

This substitution method allows the source unit to be valued as measurable unit. After that, the measurable source units are divided into header (h) and body (b) to become a transformed source unit. Header is the first line of a function source code and body is the next line of a function source code after the first line. In the Fig. 3, the lines of source unit public *boolean hasmoreelements* is the first line of function where this line is considered as the header. The rest of the lines are considered as body. This unit will become transformed source unit which is the output of this process.

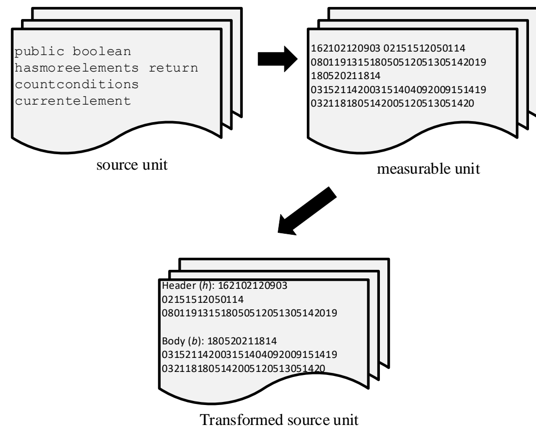

Fig. 3. Snippets of source unit is substituted and grouped to become a transformed source unit.

C. Parameterization Process

This stage will set up the parameters that would be utilized throughout the categorization process from the transformed source unit (*TSU*). The *TSUs* are in the form of transformed source unit header (*TSUh*) and transformed source unit body (*TSUb*). The parameters involved are average ratio header and average ratio body. Table I shows the parameters involved in getting the average ratio header and body and its description. In order to gain average ratio header and average ratio body, *TSU*s are calculated to gain ratio header along with ratio body. Both ratios require *TSU* to be calculated using this:

$$TSUh_n = A_1 + A_2 + A_3 + \cdots + A_n \quad (1)$$

$$TSUb_n = B_1 + B_2 + B_3 + \cdots + B_n \quad (2)$$

Since all functions have been transformed into a standardize value of public, all the source units have similar access modifier value. Therefore, these *TSU*s are divided with public access modifier weightage value (P). The P is gain through the substitution of word 'public' into162102120903 as the weightage value. The calculation of ratio header and ratio body can be visualized in equation:

$$Rh_n = \frac{TSUh_n}{P} \quad (3)$$

$$Rb_n = \frac{TSUb_n}{P} \quad (4)$$

where n = 1, 2, 3. After ratio header and ratio body is calculated, the process is tried with finding average ratio header together with average ratio body for each source unit. Ratio header and ratio body is divided with count header as well as count body respectively using this equation:

$$AVRh_n = \frac{Rh_n}{Ch_n} \quad (5)$$

$$AVRb_n = \frac{Rb_n}{Cb_n} \quad (6)$$

From here, the output of transformation process is represented in metric form as parameters.

TABLE I. DESCRIPTION OF PARAMETERS INVOLVED IN PARAMETERIZATION PROCESS

Parameters	Description
Transformed source unit header (*TSUh*)	Value of transformed source unit in header
Transformed source unit header (*TSUb*)	Value of transformed source unit in body
Ratio header (*Rh*)	The ratio of headers of the transformed source units
Ratio body (*Rb*)	The ratio of body of the transformed source units
Count header (*Ch*)	Code count of source code header in the source units
Count body (*Cb*)	Code count of source code body in the source units
average ratio header (*AVRh*)	The average ratio of header (h)
average ratio header (*AVRb*)	The average ratio of body (b)

D. Categorization Process

The categorization process occurs by comparing between two *TSU*. Assuming there are *TSUX* and *TSUY*, they are categorized into the first pool as they have the similar *AVRh*. The process of categorizing *TSU* into the first pool runs until every *TSU* with similar *AVRh* is grouped. Then, the *TSU* continue its categorization process together with the remaining from first pool to group *TSU* with similar *AVRb* into the second pool. Once the categorization process reaches its ends for the second pool, the categorization process moves to the third pool, grouping the remainder of *TSU* that cannot match to the first pool and the second pool. This process produces three pools as the output, bringing them to the final process which is match detection.

E. Match Detection Process

Finally, the pools are screened for Type-1, Type-2, Type-3, and Type-4 clones using match detection process. There are two stages of match detection for detecting code clones. The first stage is exact matching where clone Type-1 and clone Type-2 are detected from the first two pools. The match is considered as Type-1 when two average ratio header and body are identical, following the next pair of matches during the first two pool is processed. As for clone Type-2, certain pairs compared is considered as clone Type-2 when they have similar *AVRhx* and *AVRhy* but different *AVRbx* and *AVRby*, or vice versa. The second stage implements the Euclidean distance measure formula to determine the remnants from the first two pools and the third pool. Assuming the calculation involves two source units' *X* and *Y*. Each source unit consists of average ratio header and average ratio body. The formula of Euclidean distance (ED) is as follows:

$$ED = \sqrt{(AVRhX - AVRhY)^2 + (AVRbX - AVRbY)^2} \quad (7)$$

The outcome from Euclidean distance calculation is then determined its value. The value that fits within 0.85 and 1.00 is

classified as Type-3, while the remainder value is classified as Type-4.

III. Proposed Enhancement

The proposed enhancements are focusing on two processes from GCCD, which are the match detection process. The dataset for these experiments is similar to the dataset from GCCD, which is Java applications of Bellon's benchmark dataset [28]. This dataset is a benchmark in code clone domain that consists Java applications with medium size to larger size. It also provides the details to the four Java applications involved.

Previously in GCCD, the process of match detection implements Euclidean distance on to the three pools from the categorization process, in order to gain the clone Type-3 and clone Type-4. Once calculated, the value that falls between the ranges 0.85 to 1.00 is considered as clone Type-3 and suchlike is considered as clone Type-4. In comparison to previous GCCD outcomes, the aim of the first experiment is to produce a greater detection result for code clone Type-3 and code clone Type-4, by utilizing different distance measures [24]. Four distance measures were discovered for which the parameters generated from prior processes of GCCD could be applied to achieve. The distance measures are referred in Table II Manhattan distance [25], Squared Euclidean distance [26], Half-squared Euclidean distance [27], and Chi-square distance [28].

TABLE II. Equations Involved in Experiment 1

Equation Name	Equation
Manhattan distance	$d(x,y) = \sum_{i=1}^{n} \lvert x_i - y_i \rvert$
Squared Euclidean distance	$d^2(x,y) = \sum_{i=1}^{n} (x_i - y_i)^2$
Half-squared Euclidean distance	$d^2(x,y) = \sum_{i=1}^{n} (x_i - y_i)^2$
Chi-square distance	$\chi(x,y) = \sqrt{\frac{1}{2}\sum_{i=1}^{n} \frac{(x_i - y_i)^2}{(x_i + y_i)}}$

Manhattan distance calculates the absolute difference between corresponding components of two vectors, offering a simple yet effective way to assess similarity when the changes between code fragments are predominantly additive or subtractive. Squared Euclidean distance is an extension of the standard Euclidean distance, this formula squares the differences between corresponding components before summing them, placing greater emphasis on larger deviations, which may be particularly useful in distinguishing more pronounced differences in code structure. Meanwhile, half-squared Euclidean distance measures halves the squared differences, providing a balance between the sensitivity of the Squared Euclidean Distance and the simplicity of the Manhattan Distance, potentially offering a more nuanced assessment of similarity. Finally, Chi-square distance, a statistical measure that compares the observed and expected frequencies of occurrences, this formula is particularly suited for detecting differences in distributions, making it a promising candidate for identifying Type-4 clones where the code fragments may function similarly but differ significantly in their structural composition. The aforementioned equations will replace the Euclidean distance in the match detection process. The pseudocode 1 shows the pseudocode on the implementation of Chi-square distance into GCCD as an example.

Pseudocode: Match Detection Process using Chi-square distance

Pool 1, PL_1
Pool 2, PL_2
Pool 3, PL_3
Chi-Square Distance, CSD
Average ratio header, [$AVRh_1, AVRh_2, AVRh_3, \ldots AVRh_n$]
Average ratio body, [$AVRb_1, AVRb_2, AVRb_3, \ldots AVRb_n$]

1. Read [$AVRh_1, AVRh_2, AVRh_3, \ldots AVRh_n$] and [$AVRb_1, AVRb_2, AVRb_3, \ldots AVRb_n$] in PL_1 and PL_2
2. Compare $AVRh_1$ and $AVRb_1$ with $AVRh_2$ and $AVRb_2$ using exact matching technique
3. If $AVRh_1$ and $AVRb_1$ are same with $AVRh_2$ and $AVRb_2$
4. Group as Type-1
5. Else If $AVRh_1$ and $AVRh_2$ are same but $AVRb_1$ and $AVRb_2$ are different
6. Group as Type-2
7. Else If $AVRh_1$ and $AVRh_2$ are different but $AVRb_1$ and $AVRb_2$ are same
8. Group as Type-2
9. Else
10. $AVRh_1$ and $AVRh_2$ are different but $AVRb_1$ and $AVRb_2$ are different
11. Move into PL_3
12. Read remaining [$AVRh_1, AVRh_2, AVRh_3, \ldots AVRh_n$] and [$AVRb_1, AVRb_2, AVRb_3, \ldots AVRb_n$] in PL_3
13. Apply CSD between the remaining [$AVRh_1, AVRh_2, AVRh_3, \ldots AVRh_n$] and [$AVRb_1, AVRb_2, AVRb_3, \ldots AVRb_n$]
14. If distance is between 0.85 to 1.00
15. Group as Type-3
16. Else
17. Group as Type-4

By using the similar assumption from Section II(E) where there are two source units X and Y, the calculation of the match detection process can be visualized in the chi-square equation where the average ratio header and average ratio body is used:

$$CSD = \sqrt{\frac{1}{2}\left(\frac{(AVRhX - AVRhY)^2}{(AVRhX + AVRhY)} + \frac{(AVRbX - AVRbY)^2}{(AVRbX + AVRbY)}\right)} \quad (8)$$

IV. Result Analysis

The result from the experiment is recorded in two elements namely overall total clone pairs in Java applications and total clone pairs based on clone types. This section is divided into three subsections where the first subsection describes the result of overall total clone pairs and the second subsection is about the total clone pairs based on clone types, which resulted from the experiment. The final subsection analyzes and discusses the outcome from the result to pinpoint the difference between the existing GCCD and the enhancement made to GCCD.

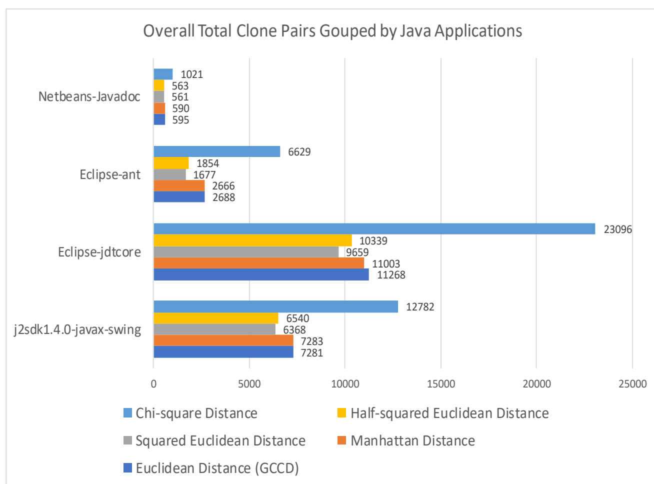

Fig. 4. Overall total clone pairs based on java applications from bellon's benchmark data.

A. *Overall Total Clone Pairs in Java Applications*

Fig. 4 illustrate the results of overall total clone pair in Java applications. The first Java application is j2sdk1.4.0-javax-swing. Based on data presented, Chi-square recorded the most total clone pair with 12782 clone pairs. The second most total clone pair is from Manhattan distance with a total of 7283 clone pairs. This puts the difference between the most clone pairs and the second-most with a total of 43.02%. Thirds go to the Euclidean distance with a total of 7281 clone pairs. This total clone pair is lower by 43.04% from the highest total clone pair by Chi-square distance. Total clone pair from Half-squared Euclidean distance recorded the fourth with a total of 6540 clone pairs. The difference between Half-squared Euclidean distance total clone pairs and the highest total clone pair is 48.83%. J2sdk1.4.0-javax-swing has lowermost total clone pair detected by Squared Euclidean distance with 6368 clone pairs. It is 50.18% less than the highest total clone pair by Chi-square distance.

The second Java application for detecting overall total clone pair is Eclipse-jdtcore. The top result for total clone pair is by the Chi-square distance with a total of 23096 clone pairs. The next in order is by Euclidean distance with a total of 11268 clone pairs. The gap between the highest and the second-highest overall total clone pair is 51.21%. Manhattan provides the third-most total clone pair with 11003 clone pairs. This is 52.36% lower than the highest overall total clone pair detected in Eclipse-jdtcore. The fourth result of total clone pair is Half-squared Euclidean distance with 10339 clone pairs. The difference between overall total clone pair from Half-squared Euclidean distance with the most total clone pair from Chi-square distance is 55.23%. The last from Eclipse-jdtcore is by Squared Euclidean distance with a 9659 total of clone pairs. This put the difference between the lowest and the highest from Eclipse-jdtcore with a difference of 58.18%.

The third Java application is Eclipse-ant. Overall total clone pair by Chi-square distance recorded the highest value with 6629 clone pairs. The second is followed by Euclidean distance where it gained a total of 2688 clone pairs, which left the gap of 59.45%. The third for overall total clone pair of Eclipse-ant is Manhattan distance which turned in a total of 2666 clone pairs. This put a 59.78% difference between the first and the third overall total of clone pair in Eclipse-ant. The fourth for Eclipse-ant is Half-Squared Euclidean distance, which has a total of 1854 clone pairs. This is 72.03% lower than the highest overall total clone pair in Eclipse-ant, which is the Chi-square distance. The last formula that detects a total of 1677 clone pairs is Squared Euclidean distance. This has made a 74.70% gap with Chi-square distance, the highest for Eclipse-ant.

As for Netbeans-javadoc, the most prominent total clone pair is by Chi-square distance with 1021 clone pairs. The subsequent total clone pair is by Euclidean distance. It recorded a total of 595 clone pairs, leaving the gap of 41.72% lower than the highest overall total clone pair result in Netbeans-javadoc. Next, Manhattan distance set a total of 590 clone pairs. This overall total clone pair value is lower than the highest value which is 42.21% difference. The fourth total of clone pairs is Half-squared Euclidean distance with a total of 563 clone pairs. The difference between the fourth and the first value of overall total clone pair in Netbeans-javadoc is 44.86%. 561 clone pairs are the final overall total clone pair by Squared Euclidean distance. The difference between the least and the most overall total clone pair detected in Netbeans-javadoc is 45.05%.

B. *Total Clone Pairs Based on Clone Types*

This part of the result is discussed based on each Java applications by Bellon's benchmark data, based on Table III.

1) j2sdk1.4.0-javax-swing: Table III depicted the result of total clone pairs for each Java application. In j2sdk1.4.0-javax-swing, the detected clone pairs Type-1 is by Half-Squared Euclidean distance with 892 clone pairs. The second highest value of detecting clone Type-1 is by Chi-square distance, with 891 clone pairs, preceding about 0.11% difference from the highest clone pair Type-1 value. The third highest is by Manhattan distance with 889 clone pairs. The gap difference from the highest value of clone pair Type-1 detected in j2sdk1.4.0-javax-swing is 0.34%. The fourth value of clone pair Type-1 detected in this application is 888 clone pairs by Squared Euclidean distance, with 0.45% gap difference. Finally, Euclidean distance recorded the lowest value of clone Type-1 detected by 877 clone pairs, leaving percentage difference of 1.68%. Code clone Type-2 detection clone pair for Half-Squared Euclidean distance keep as highest value recorded with 3725 clone pairs. It is then followed by Manhattan distance with 3716 clone pairs, leaving a percentage difference of 0.24%. The third most value of clone pair Type-2 detected in j2sdk1.4.0-javax-swing is by Euclidean distance with 3697 clone pairs. The percentage difference between Euclidean distance's value and the highest value of clone pair Type-2 is 0.75%. The fourth value is by Squared Euclidean distance with 3695 clone pairs and percentage difference of 0.81%. The least clone pair Type-2 detected is by Chi-square distance with 3684 clone pairs. The percentage difference is 1.10%. Meanwhile, Chi-square distance gained the highest total clone pair Type-3 with 3633 clone pairs. This was followed by Half-squared Euclidean distance with a value of 1773 clone pairs. It is 51.20% less

than the Chi-Square distance result. The third and fourth total clone pairs Type-3 are Manhattan distance (1727 clone pairs) as well as Squared Euclidean distance (1718 clone pairs), with each gap difference of 52.46% and 52.71% respectively. The lowest total clone pair Type-3 is by Euclidean distance with a total of 1710 clone pairs, leaving a 52.93% lower than Chi-square distance. Next, Chi-square Distance was the highest in j2sdk1.4.0-javax-swing code clone Type-4, with a value of 4574 clone pairs. The Euclidean distance, which detected 997 clone pairs, is the second highest for clone Type-4. It is 78.20% lower than the highest value of clone Type-4 detected. Manhattan distance is the thirds with 951 clone pairs of Type-4, a 79.21% lower than Chi-square distance. The fourth value is by Half-squared distance with 150 clone pairs (96.72%) and the least value is from Squared Euclidean distance with a value of 67 clone pairs (98.54%) for clone Type-4.

TABLE III. TOTAL CLONE PAIRS BASED ON CLONE TYPES FOR EACH DISTANCE MEASURES

Bellon's Benchmark Data Java Applications	**Clone Type**	**Total clone pairs based on clone types**				
		ED* ***(GCCD)***	***MD****	***SED****	***HSED****	***CSD****
j2sdk1.4.0-javax-swing	T-1*	877	889	888	892	891
	T-2*	3697	3716	3695	3725	3684
	T-3*	1710	1727	1718	1773	3633
	T-4*	997	951	67	150	4574
Eclipse-jdtcore	T-1*	626	627	627	627	627
	T-2*	2886	2884	2887	2887	2886
	T-3*	4265	3880	3782	4564	7576
	T-4*	3491	3612	2363	2261	12007
Eclipse-ant	T-1*	185	185	185	185	185
	T-2*	552	650	650	650	650
	T-3*	581	562	535	585	2061
	T-4*	1370	1269	307	434	3733
Netbeans-javadoc	T-1*	99	99	99	99	99
	T-2*	341	338	338	338	338
	T-3*	102	102	102	104	197
	T-4*	53	51	22	23	387

a. *T-1 = clone Type-1, T-2 = clone Type-2, T-3 = clone Type-3, T-4 = clone Type-4, ED = Euclidean Distance, MD = Manhattan Distance, SED = Square Euclidean Distance, HSED = Half-squared Euclidean Distance, CSD = Chi-square Distance

2) Eclipse-jdtcore: Moving forward with the second Java application, which is Eclipse-jdtcore. The overall total clone pair Type-1 detected is consistent for each distance measure with a value of 627 clone pairs. The exception is from Euclidean distance, which detected 626 clone pairs, leaving a 0.16% gap difference from other distance measure formula. In regards to clone Type-2, the highest value went to Squared Euclidean distance as well as Half-Squared Euclidean distance, with each distance measure, scored a total clone pairs of 2887. The second greatest value for Type-2 clone is by Chi-square distance and Euclidean distance (2886 clone pairs). The lowest value is by Manhattan distance (2884 clone pairs). Both with the percentage difference of 0.03% and 0.10% from the highest value. Next, the Chi-square distance gained the highest value of 7576 clone pairs Type-3 for Eclipse-jdtcore. Half-squared Euclidean distance recorded the second-highest Type-3 value of 4564 clone pairs. It is 39.76% lower than Chi-square distance. The next distance measure followed is Euclidean distance with 4265 clone pairs of Type-3, marking a 43.70% gap from Chi-square distance. The fourth and the lowest Type-3 clones are Manhattan distance (3880 clone pairs) together with Squared Euclidean distance (3782 clone pairs). The Manhattan distance and Squared Euclidean distance are 48.79% as well as 50.08% lower than Chi-square distance. Then, clone detection for Type-4 by Chi-square distance in Eclipse-jdtcore, is the highest with a total of 12007 clone pairs. This is followed by the second highest with a value of 3612 clone pairs Type-4 by using Manhattan distance. It is 69.92% lower than Chi-square distance. The third and fourth for Type-4 in Eclipse-jdtcore are recorded by Euclidean distance (3491 clone pairs) along with Squared Euclidean distance (2363 clone pairs). They have 70.93% and 80.32% lower than Chi-square distance. The lowest value of clone pair Type-4 for Eclipse-jdtcore is by Half-squared Euclidean distance, with 2261 total clone pairs and 81.17% gap difference.

3) Eclipse-ant: Total clone pairs Type-1 in Eclipse-ant showed concordant outcomes for every experimented distance measure which is 185 clone pairs along individually. For Type-2, the highest clone pair value detected is by each distance measure with 650 clone pairs, except for Euclidean distance with 552 clone pairs, which is 15.08% lower than the highest value. Meanwhile, Chi-square distance showed the highest total of 2061 clone pairs Type-3, followed by Half-squared distance with a total of 585 clone pairs. Half-squared distance has 71.62% lower than Chi-square distance. The third value for clone Type-3 is by Euclidean distance with a total of 581 clone pairs and 71.81% gap difference from highest value. The fourth value is by Manhattan distance with 562 clone pairs. It is 72.73% lower than Chi-square distance. The lowest value for Type-3 clones in Eclipse-ant is Squared Euclidean distance with 535 clone pairs and 74.04% lower than Chi-square distance. Next, the Chi-square distance for clone Type-4 in Eclipse-ant has the greatest value of 3733 clone pairs. The second-highest total clone pairs are 1370 clone pairs Type-4 by Euclidean distance, with 63.30% lower than Chi-square

distance. The third and fourth values are shown by Manhattan distance (1269 clone pairs) as well as Half-squared Euclidean distance (434 clone pairs). Both has 66.01% and 88.37% lower than Chi-square distance. The least value of total clone pairs Type-4 in Eclipse-ant is by Squared Euclidean distance with 307 clone pairs. It is 91.78% lower than the highest value by Chi-square distance.

4) Netbeans-javadoc: The Netbeans-javadoc application was also revealed to have concordant outputs when it comes to clone Type-1 for each distance measure which is 99 clone pairs. Euclidean distance recorded the highest Type-2 value with 341 clone pairs. Other distance measures detected 338 clone pairs, 0.88% lower than Euclidean distance. For Type-3, Chi-square distance recorded the highest with 197 clone pairs, followed by Half-squared distance with 104 clone pairs. It is 47.21% lower than Chi-square distance. Euclidean distance, Manhattan distance as well as Squared Euclidean distance showed the lowest which is 102 clone pairs. It has 48.22% lower than Chi-square distance. For Type-4 in Netbeans-javadoc, the Chi-square distance also showed the highest value with 387 total clone pairs. The second highest is by Euclidean distance which is 53 clone pairs (86.30%) of Type-4, followed by the third value by Manhattan distance which is 51 clone pairs (86.82%). The fourth value is 23 clone pairs by Half-squared distance and the lowest Type-4 clone value is 22 clone pairs by Squared Euclidean distance. Both distances have 94.06% and 94.32% gap difference from Chi-square distance.

V. Discussion

The experiment is concentrating on enhancing match detection process of GCCD by substituting the Euclidean distance to different distance measures. The enhancement on match detection process should be affecting the detection result on clone Type-3 and Type-4, as clone pairs is calculated using the distance measure formula. In this experiment, Chi-square distance has shown a significant increase on the overall total clone pairs that is detected in Java applications of Bellon's benchmark data. Moreover, Chi-square distance shows an improvement in total clone pairs based on clone types, which detected the highest value for each clone Type-3 as well as Type-4 in Eclipse-ant and Netbeans-javadoc application, respectively. Chi-square distance managed to keep the similar value of total clone pairs Type-1 other distance measures in the Eclipse-jdtcore, Eclipse-ant and Netbeans-javadoc. Chi-square distance also able to maintain the similar clone pairs Type-2 value with other distance measures for Eclipse-ant and Netbeans-javadoc. However, Chi-Square is placed as the second highest total clone pairs based on clone Type-1 and the least value in j2sdk1.4.0 – javax-swing. Another difference is in Eclipse-jdtcore application, where Chi-square distance detected the second highest value of clone pair Type-2. Based on this experiment, Euclidean distance and Chi-square distance both embody the similar structure formula. Nonetheless, Chi-square distance divides the upper value with a frequency inverse using the weightage summation of both header and body. As for the difference in result on clone Type-1 and Type-2 in two of the mentioned Java applications, runtime performance during the pre-processing process and transformation process might have affected the detection result. Thus, Experiment 1 concludes that the implementation of Chi-square distance increases the clone pair detection result specifically in Type-3 and Type-4, respectively.

VI. Conclusion

In this paper, we introduced an improvement that is to the GCCD for detecting code clones in each clone type, specifically Type-3 and Type-4, due to the earlier result from GCCD implicit the inconsistency of the clone detection result in Java applications in Bellon's benchmark dataset. The enhancement that proposed the detection result of clone pair Type-3 and Type-4 can be improved by enhancing match detection process of GCCD, through modifying the distance measures. Result from the experiment revealed that the implementation of Chi-Square provides a higher code clone detection result as the GCCD is enhanced using Chi-square distance in match detection process. The improvement can be seen specifically when it has overruled GCCD by detecting the highest clone pairs Type-3 and Type-4.

The model currently supports clone detection within Java applications as the dataset is limited to the Bellon's benchmark dataset Java application. As a future enhancement, experiment on detecting and analyzing clone pairs in Python applications will be conducted.

Acknowledgment

The authors would also like to thank the Malaysian Higher Education Ministry and Universiti Malaysia Pahang Al-Sultan Abdullah for their support for this project through the Fundamental Research Grant Scheme (FRGS Grant ID: FRGS/1/2024/ICT01/UMP/02/1).

References

[1] B. Van Bladel and S. Demeyer, "A Comparative Study of Code Clone Genealogies in Test Code and Production Code," Proc. - 2023 IEEE Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2023, pp. 913–920, 2023, doi: 10.1109/SANER56733.2023.00110.

[2] M. Nashaat, R. Amin, A. H. Eid, and R. F. Abdel-Kader, "An enhanced transformer-based framework for interpretable code clone detection," J. Syst. Softw., vol. 222, p. 112347, Apr. 2025, doi: 10.1016/J.JSS.2025.112347.

[3] B. Hu, D. Yu, Y. Wu, T. Hu, and Y. Cai, "An empirical study of code clones: Density, entropy, and patterns," Sci. Comput. Program., vol. 242, p. 103259, May 2025, doi: 10.1016/J.SCICO.2024.103259.

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

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

[6] N. Saini, S. Singh, and Suman, "Code Clones: Detection and Management," in Procedia Computer Science, Jan. 2018, vol. 132, pp. 718–727, doi: 10.1016/j.procs.2018.05.080.

[7] YuHao, HuXing, LiGe, LiYing, WangQianxiang, and XieTao, "Assessing and Improving an Evaluation Dataset for Detecting Semantic Code Clones via Deep Learning," ACM Trans. Softw. Eng. Methodol., Jul. 2022, doi: 10.1145/3502852.

[8] Z. Zhang and T. Saber, "Assessing the Code Clone Detection Capability of Large Language Models," ICCQ 2024 - Proc. 4th Int. Conf. Code Qual., pp. 75–83, 2024, doi: 10.1109/ICCQ60895.2024.10576803.

[9] N. N. Mokhtar, A.-F. Mubarak-Ali, and M. A. Mohamad Hamza, “Enhanced Pre-processing and Parameterization Process of Generic Code Clone Detection Model for Clones in Java Applications,” IJACSA) Int. J. Adv. Comput. Sci. Appl., vol. 11, no. 6, 2020, Accessed: Jun. 06, 2021. [Online]. Available: www.ijacsa.thesai.org.

[10] C. Tao, Q. Zhan, X. Hu, and X. Xia, “C4: Contrastive Cross-Language Code Clone Detection,” IEEE Int. Conf. Progr. Compr., vol. 2022-March, pp. 413–424, 2022, doi: 10.1145/3524610.3527911.

[11] A. F. Mubarak-Ali and S. Sulaiman, “Generic Code Clone Detection Model for Java Applications,” in IOP Conference Series: Materials Science and Engineering, Jun. 2020, vol. 769, no. 1, doi: 10.1088/1757-899X/769/1/012023.

[12] J. Martinez-Gil, “Source Code Clone Detection Using Unsupervised Similarity Measures,” Lect. Notes Bus. Inf. Process., vol. 505 LNBIP, pp. 21–37, Jan. 2024, doi: 10.1007/978-3-031-56281-5_2.

[13] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach,” Sci. Comput. Program., vol. 74, no. 7, pp. 470–495, May 2009, doi: 10.1016/j.scico.2009.02.007.

[14] G. Shobha, A. Rana, V. Kansal, and S. Tanwar, “Comparison between Code Clone Detection and Model Clone Detection,” 2021 9th Int. Conf. Reliab. Infocom Technol. Optim. (Trends Futur. Dir. ICRITO 2021, 2021, doi: 10.1109/ICRITO51393.2021.9596454.

[15] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, “A Systematic Review on Code Clone Detection,” IEEE Access, vol. 7. Institute of Electrical and Electronics Engineers Inc., pp. 86121–86144, 2019, doi: 10.1109/ACCESS.2019.2918202.

[16] C. K. Roy and J. R. Cordy, “An empirical study of function clones in open source software,” in Proceedings - Working Conference on Reverse Engineering, WCRE, 2008, pp. 81–90, doi: 10.1109/WCRE.2008.54.

[17] M. Mondal, C. K. Roy, and J. R. Cordy, “NiCad: A Modern Clone Detector,” Code Clone Anal., pp. 45–50, 2021, doi: 10.1007/978-981-16-1927-4_3.

[18] W. Wang, Z. Deng, Y. Xue, and Y. Xu, “CCStokener: Fast yet accurate code clone detection with semantic token,” J. Syst. Softw., vol. 199, p. 111618, May 2023, doi: 10.1016/J.JSS.2023.111618.

[19] S. Giesecke, “Generic Modelling of Code Clones,” Duplic. Redundancy, Similarity Softw., no. 06301, pp. 1–23, 2007, [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/960.

[20] B. Biegel and S. Diehl, “Highly Configurable and Extensible Code Clone Detection,” in Proceedings - Working Conference on Reverse Engineering, WCRE, 2010, pp. 237–241, doi: 10.1109/WCRE.2010.34.

[21] C. J. Kapser, J. Harder, and I. Baxter, “A Common Conceptual Model for Clone Detection Results,” in 2012 6th International Workshop on Software Clones, IWSC 2012 - Proceedings, 2012, pp. 72–73, doi: 10.1109/IWSC.2012.6227870.

[22] A.-F. Mubarak Ali, S. Sulaiman, and S. M. Syed-Mohamad, “An Enhanced Generic Pipeline Model for Code Clone Detection ,” 2011, Accessed: Jun. 06, 2021. [Online]. Available: https://ieeexplore-ieee-org.ezproxy.ump.edu.my/document/6140712.

[23] N. S. Zaidi, A. F. Mubarak-Ali, A. S. Fakhrudin, and R. N. Romli, “Determining the Best Weightage Feature in Parameterization Process of GCCD Model for Clone Detection in C-Based Applications,” 8th Int. Conf. Softw. Eng. Comput. Syst. ICSECS 2023, pp. 280–285, 2023, doi: 10.1109/ICSECS58457.2023.10256395.

[24] N. A. Abdullah, M. Azwan Mohamad Hamza, and A. F. M. Ali, “A Review on Distance Measure Formula for Enhancing Match Detection Process of Generic Code Clone Detection Model in Java Application,” Proc. - 2021 Int. Conf. Softw. Eng. Comput. Syst. 4th Int. Conf. Comput. Sci. Inf. Manag. ICSECS-ICOCSIM 2021, pp. 285–290, Aug. 2021, doi: 10.1109/ICSECS52883.2021.00058.

[25] A. Muhammad et al., “Distance Measurements Method for the Demite Pronunciation Assessment,” ICSET 2018 - 2018 IEEE 8th Int. Conf. Syst. Eng. Technol. Proc., pp. 189–194, Jan. 2019, doi: 10.1109/ICSENGT.2018.8606375.

[26] A. Kazemi, S. Sahay, A. Saxena, M. M. Sharifi, M. Niemier, and X. S. Hu, “A Flash-Based Multi-Bit Content-Addressable Memory with Euclidean Squared Distance,” 2021 IEEE/ACM Int. Symp. Low Power Electron. Des., pp. 1–6, Jul. 2021, doi: 10.1109/ISLPED52811.2021.9502488.

[27] TIBCO Software Inc., “Square Euclidean Distance and Half Square Euclidean Distance,” stn.spotfire.com, 2012. https://docs.tibco.com/pub/spotfire/7.0.0/doc/html/hc/hc_square_half_square_euclidean_distance.htm (accessed Aug. 26, 2021).

[28] M. Majhi and A. K. Pal, “An image retrieval scheme based on block level hybrid dct-svd fused features,” Multimed. Tools Appl., vol. 80, no. 5, pp. 7271–7312, Oct. 2021, doi: 10.1007/s11042-020-10005-5.