INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING & COMPUTER SYSTEMS (IJSECS)
ISSN: 2289-8522 e-ISSN: 2180-0650
VOL. 8, ISSUE 1, 1 – 10
DOI: https://doi.org/10.15282/ijsecs.8.1.2022.1.0091

ORIGINAL ARTICLE

Enhancement of Generic Code Clone Detection Model for Python Application

Ilyana Najwa Aiza Asmad[1], Al-Fahim Mubarak Ali[2], Nik Intan Syahiddatul Ilani Jailani[3]

[1,2,3]Faculty of Computing, College of Computing and Applied Sciences, Universiti Malaysia Pahang, 26600 Pekan, Pahang Darul Makmur

ABSTRACT – Identical code fragments in different locations are recognized as code clones. There are four native terminologies of code clones concluded as Type-1, Type-2, Type-3 and Type-4. Code clones can be identified using various approaches and models. Generic Code Clone Detection (GCCD) model was created to detect all four terminologies of code clones through five processes. A prototype has been developed to detect code clones in Java programming language that starts with Pre-processing Transformation, Parameterization, Categorization and ends with the Match Detection process. Hence, this work targeted to enhance the prototype using a GCCD model to identify all clone types in Python language. Enhancements are done in the Pre-processing process and parameterization process of the GCCD model to fit the Python language criteria. Results are improved by finding the best constant value and suitable weightage according to Python language. Proposed enhancement results of the Python language clone detection in GCCD model imply that Public as the weightage indicator and def as the best constant value.

ARTICLE HISTORY
Received: 11 May 2021
Revised: 28 June 2021
Accepted: 7 Oct 2021

KEYWORDS
Code Clone Detection
Python Languages
Computational Intelligence

INTRODUCTION

Repeated usability of a source in a software system is known as code clone [1]. On the bright side, this practice is proven to speed up code writing in the development phase [2]. However, code reusability is classified as a limitation that will diverge within the enhancement of the software that mostly will affect the software maintenance process [3]. Multiple types of research found 22.3% of Linux clones in large software systems [4] and 29% of clones detected in JDK [5]. Only less than 5% to 10% of duplicated clone portions are somehow considered healthy [6] [7].

Code clones have four common native code clone terminologies that were agreed upon by [8]. Those terminologies are Type-1, Type-2, Type-3 and Type-4. Type-1 is classified as 100% identical copied code from the original one. Type-2 is code that contains similar fragments with few additions of identifiers. Type-3 is where codes are additionally modified from Type-2 which involves changing, adding, or removing statements. Type-4 is where codes are semantically modified.

Clone detection varies within six major comparison approaches at different granularity to transform and identify code clones. The six comparison approaches are text-based, token-based, tree-based, metric-based, and graph-based and hybrid comparison approaches. Text-based approaches compare strings in source codes using the 'AND' statement. The token approach transforms lines of source code into tokens and compares the sequence of tokens accordingly [9]. The graph-based approach uses the graph technique to compare the codes with the aid of Program Dependence Graph (PDG) [10]. Metric based approach works by using code fragments metrics to perform a comparison with each other. The tree-based approach compares assigned hash functions to partition subtrees of the tree. The last approach which is hybrid based is a combination of two or more different approaches used to detect clones.

Python language is widely used in recent times and ranked up as the 4th most-used language in GitHub [11]. Many recent applications are utilizing Python as their programming language. An experiment was conducted to ensure the clone's existence in Python applications. The conducted experiment uses two applications and results in the existence of clones with a total number of 2051 [12]. Therefore, the occurrences of clones in Python language applications are inevitable.

The few biggest drawbacks of repeated codes can lead to a significant increase in maintenance cost as code duplication increases duplicate bugs and bad designs [13] [14]. Findings of code cloning in various programming languages agree that cloned code appears to be more unstable than original codes [3]. Generic Code Clone Detection model [14]. It is necessary to create a suitable clone detector for Python programming language that covers clone detection until Type-4 by enhancing the pre-processing and the transformation of the source codes that can improve the representation of the source units for code clone detection for Python language.

The remainder of this paper is organized as follows: Section 2 briefly presents works related to code clone detection models. Section 3 describes the Generic Code Clone Detection in brief. Section 4 provides the proposed enhancement for the Generic Code Clone Detection in Python application while Section 5 shows the experimental result of the enhancement. Section 5 discusses the experimental results and Section 6 concludes the paper.

*CORRESPONDING AUTHOR | A. F. Mubarak | ✉ fahim@ump.edu.my

RELATED WORK

Four existing models were created to detect clones in a unified pattern. The four models are the Generic Clone Model, Generic Pipeline Model, Unified Clone Model, and Generic Code Clone Detection Model. Generic Clone Model highlights more on layering the classification on detection of the clones, explanation of the clone, and management of the clone [15]. Generic Pipeline Model consists of five processes that start~~s~~ with a parsing process, continue with the pre-processing process, pooling process, comparing process, and ends up with filtering process [16]. Unified Clone Model is an analysis that manages to produce four groups of outcomes which are responsible for clone management and triage, further integration of data from other sources, replicas of other scientific findings, and evaluation from different clones detection techniques [17]. Generic Code Clone Detection Model involves five processing stages that start with pre-processing the source code, code transformation process, parameterization, categorization, and finish off with a match detection process [14].

Generic Clone model was proposed in 2006 to identify available clones in a software program [15]. This model alternately represents clone detection, clone details, clone management, and clone removal in separate layers. By representing models in layers, this model intentionally eases the implementation process in tools that support the principle of clone detection, clone definition, clone management, and clone removal. Any tools that support all four functions where it can detect clones, define the type of clones, manage the clones, and remove clones are taking advantage of this Generic Clone Model.

Generic Pipeline Model starts with the Parsing process where this process is responsible for transforming source files into source units. This model adopts tree base approaches since the source units in this model are presented as sub-tree as in AST. The source unit will then be normalized according to the format of regular form in the Pre-Processing process to produce Pre-Processing source file in AST. The Pre-Processing AST will go through a Pooling process where similar types of AST characteristics are gathered to form a group or set that is known as pool data. Pools data will be compared to each other in a comparison process to define a similarity group. The similarity group will then be filtered in the final Filtering process where all unwanted sets of cloned data.

The unified clone Model is a clone detection model that is still in ongoing research conducted by [17]. This model attempts to develop a generic approach that will represent the result of all code clone detection tools. Unified Clone Model is documented only in the design phase. The outcome of concept analysis conducted within eleven application use cases has been grouped under four categories that include detection for clone triage and management, integration of data from other sources, replicas of other research, and benchmarking of code clone detection techniques. This model is still in the design phase and lacks data documentation with further information [18].

The first process in Generic Code Clone Detection Model is Pre-Processing process where the Source code will be divided into fragments that are also known as source units. In this process, parts of unnecessary codes such as comments, empty lines, will be removed. Regularization of function access and all text in codes are converted into lowercase also occur during Pre-Processing process. The transformation process is where the Source unit from the previous process will be transformed into a numeric pre-processed unit so that the result of the Generic Code Clone Detection model can be counted and measured. Numeric source units are classified into two groups which are either in header (h) or body (b). The parameterization process is where numeric source units were taken to be processed into metrics value. The metrics value then will be measured and gathered in this process to form sets of pools. A set of pools are then used as input for the last process. Match Detection process. The last process will compare the sets of pools from the previous process to get the result of code clone detection. The result of code clone detection in this model will classify all types of code clone terminologies from Type-1 until Type-4. Table 1 shows the SWOT analysis of related code clone detection models.

Table 1. Code Clone Detection Model SWOT Analysis

Features	Generic Clone model [15]	Generic Pipeline Model [16]	Unified Clone Model [17]	Generic Code Clone Detection Model [14]
Strength	-Layered concept that eases the effort in implementation algorithm within tools that supports the principle of generic clone detection, clone definition, clone management, and clone removal. -Support different algorithms within the same generic principle.	-Step by step process in this model allows better customization to be made. -Some process can be altered or modified, while some of the other allowed to be used as in general setting.	-Designed according to accommodate various research in the code clone detection field. -Result will be in the various desired output according to the principle used in specific clone detection.	-Use numerical source units that aid in the Analysis phase. Can detect all types of clone terminologies. -Processes are linear and ~~in~~ step by step which allows better customization and modification to be made to the model's processes.

Weakness	-The layers' structures and functions are fixed where there is no room for alteration to the layers. -Earliest model in the clone detection field where the model focuses more on fundamentals and lacking in allowing changes.	-Manipulation on pre-defined rules and sets results in Limited extension of this model. -Limited access to result analysis due to no numerical data are used within the processes that limit the ability to measure the effectiveness.	-Lack of documented information due to still under design phase. -There were no available tools that have been using this model to guide on to more research and findings.	-Current prototypes can only suit Java and .Net programming language. -Alteration needs to be made with the skills within the area of Java language only.
Opportunity	-The clone description layers can be improved. A comparison algorithm can be created with variety.	-The models can be improved to generate better clone results. -Changes could be made with different approaches within every process to ease the result comparison.	-User-defined process concepts were open to more ideas. The new concept can be added and welcomed to the model principle.	-Prototype can be improved to be used as clone detection tools for other programming languages. -More research can be done to indicate the numerical concept ideas of this model.
Threat	-Plugins tool of this model is impossible to be altered.	-Evaluated model will end up with different results than results produced by existing tools.	-Eleven different tool concepts will lead to various results.	-Prototypes with large sizes that require good performance computers to run on.

GENERIC CODE CLONE DETECTION MODEL

Generic Code Clone Detection model was built alongside with a prototype aims to detect clones and clone types in Java language application [14]. The five processes that involved in this model were illustrated as in Figure 1 with the input and output of each process.

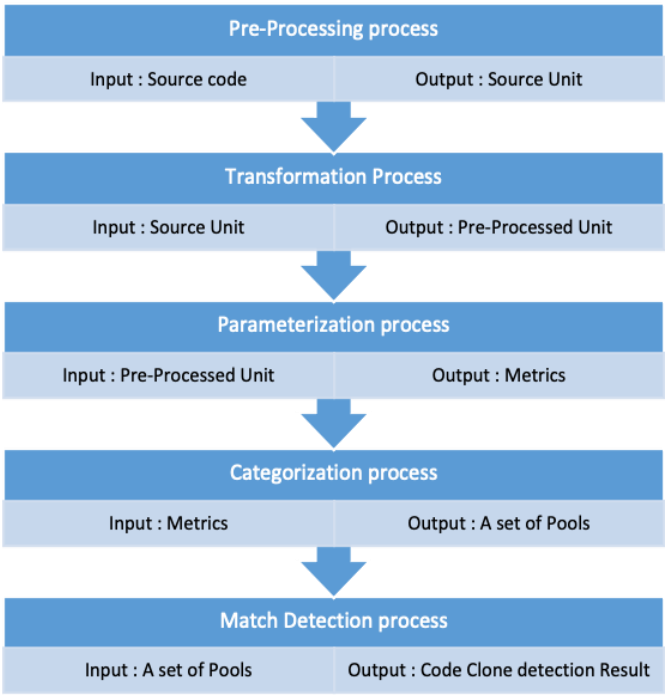

Figure 1. Generic Code Clone Detection Model

Pre-processing Process

Detecting clones by using this model will undergo the first process which is Pre-processing process where source codes as the input will be divided and processed into source units. There are five rules required for this process to produce a source unit. Source units are simply the fragments of source codes that have been standardized. Table 2 shows the five rules within the pre-processing process.

Table 2. Pre-Processing rules.

Pre-Processing Rules	Rules Details
PPR-1	Exclude source codes import lines and package lines.
PPR-2	Exclude source codes comment lines.
PPR-3	Exclude source codes null statements.
PPR-4	Replace function modifier with public.
PPR-5	All source codes regularized with lowercase.

These five rules will be further explained with the example of function A that was written in Python language :
def myIntro():

print(“Simple Python”)

The output of pre-processing process of function A will be as :

public myintro printsimple python

Transformation process

The transformation phase will transforms smaller units of source code into comparable properties that fit the detection models or approaches. Source units from the pre-processing process will either be transformed into tokens, or trees, or metrics according to detection models and approaches. For this model, the source unit will then be transformed into the numerical form of a pre-processed unit that can be measured in the next process. The core idea of the transformation process in this model is to substitute the code’s letters into numbers according to the location of alphabetical order. As an example, the word cd will be transformed into the number 0304 due to c is located in the third order in the alphabet list, and d is in the fourth.

The numerical form then will be classified into two groups which are either in the header (h) part of the code or body (b) part of the code. As an example, function A output from the pre-processing process will be classified into the header (h) or body (b) in the transformation process as followed :

Header (h) : public myintro

Body (b) : printsimple python

Parameterization process

Header and body from transformation process will undergo parameterization process to define the metrics that will be then used in the next process. The average ratio of the header and body are the main parameter involved in this model. There are four main metrics taken from the output of the transformation process to get the value of the average header and body ratio. Table 3 shows the list of metrics taken from the information of header and body.

Table 3. Metric that are taken from header and body.

Metrics	Information
Header value	Total value of codes that are counted in header (h).
Body value	Total value of codes that are counted in body (b).
Header ratio	Ratio of the header (h) according to header and body value.
Body ratio	Ratio of the body (b) according to the header and body value.
Average header ratio	Calculated average header (h) ratio.
Average body ratio	Calculated average body (b) ratio.

All functions modifiers hold the same access modifier value since it has been replaced with public according to the PPR-5 in the initial process. For instance, the average header and body ratio are the result of the numerical source unit divided by the value of the public access modifier. The numerical source unit for the header will be defined as *NSUh* and *NSUb* for the body. Hence, the calculation of the involved metrics to get the average ratio for both header and body will be as follows :

$$HR = \frac{(H1,H2,H3 \ldots Hn)}{P} \quad (1)$$

$$BR = \frac{(B1,B2,B3 \ldots Bn)}{P} \quad (2)$$

$$AHR = \frac{HR}{HV} \quad (3)$$

$$ABR = \frac{BR}{BV} \quad (4)$$

1 and 2 are referring to the calculation of header and body ratio. where;
P stands for the standard value that public modifier hold.
HR stands for header ratio.
BR stands for body ratio.
H1,H2,H3…Hn equals to the numerical source unit for the header.
B1,B2,B3…Bn equals to the numerical source unit for the body.
While 3 qnd 4 are referring to the calculation of average header and body ratio. Where:
AHR holds value of the average header ratio.
ABR holds value of the average body ratio.
HV equals to header value.
BV equals to body value.

This parametrization process is mainly to produce the listed metrics in Table 3 as the output from this process to be used as input for the upcoming process.

Categorization Process

Numerical source units of functions then will be pooled in groups according to the calculated average header and body ratio from the previous process. There will be the exact three groups in the pooling process.

Functions with different numerical source units that hold the same header value will be in the first pool. For example, if the numerical source unit of function X and Y hold the same value of the average header ratio, those two functions will be in the same pool. While functions with different numerical source units that hold the same body value will be in the second pool. For example, if the numerical source unit of functions A and B hold the same value of average body ratio, those two functions will be in the same pool. The numerical source unit of other functions that do not belong to those two pools will then be in the third pool.

Match Detection Pocess

All the numerical source code units will be matched to identify the clone types which are Type-1, Type-2, Type-3, or Type-4. This model match detection methods are rules by using an exact matching comparison algorithm in detecting Type-1 and Type-2 while the Euclidean distance comparison algorithm is applied for detecting Type-3 and Type-4. The exact matching initially will take place in detecting clones from first and second pools from the previous process, those numerical source units that do not match the Type-1 and Type-2 will be gathered together with the third pool to be matched using Euclidean distance to detect clones with Type-3 and Type-4.

The detection of Type-3 and Type-4 using Euclidean distance algorithm, E, works as calculation below with the example of two numerical source unit of function X and Y :

$$\mathrm{EXY} = (ahrX - ahrY)^2 + (abrX - abrY)^2 \quad (5)$$

Which;

EXY is the value of Euclidean distance of numerical source unit for function X and Y.
ahrX holds value of average header ratio for function X.
ahrY holds value of average header ratio for function Y.
abrX holds value of average body ratio for function X.
abrY holds value of average body ratio for function Y.

The Type-3 are detected if the Euclidean distance holds the value between 0.85 to 1, while Type-4 are detected if the value it holds is not in the range of 0.85 to 1.

PROPOSED ENHANCEMENT

Generic Code Clone Detection Model for Python language is proposed to detect code clones type in Python language applications using Generic Code Clone Detection model (GCCD). Enhancements that have been made by using this model involved in Pre-processing process and Parameterization process.

Enhancement in Pre-processing Process

The enhancement aim of pre-processing process in this model is to standardize the original source codes into source units that are better fit to this model according to Python language syntax. The tweak involved in this pre-processing model is stated in the fourth rule, PPR-4 that responsible for replacing and standardize the Python language function modifier that was originally def into public. Hence, the other four rules are remaining the same for this enhanced model. The enhanced pseudo-code of this process is shown in Figure 2.

```
Python application, Py1
python application source file, [F1,F2,F3,..Fn]
python application source code, [C1,C2,C3,..Cn]
python application source unit, [U1,U2,U3,..Un]
PPR-1
PPR-2
PPR-3
PPR-4
PPR-5

1. Read source file F1 in Py1
2. Every F1,
3. Check C1,
4.      Every C1
5.        Commit PPR-1
6.        Commit PPR-2
7.        Commit PPR-3
8.        Commit PPR-4
9.        Commit PPR-5
10.       Repeat for the next source codes [C2,C3,..Cn]
          of the F1.
11. Repeat step 2 to 10 for the next available
    source codes [F2,F3,..Fn] od the Py1S
```

Figure 2. Pseudo code for Pre-processing process.

Enhancement in Parameterization Process

The enhancement done in the parameterization process aims to produce metric parameters by changing the value of the function access modifier from def to the public that holds the weightage of 162102120903. The value of weightage is determined by the numerical location order of the alphabet sequence as stated in the Transformation process. The enhanced pseudo code for this process is shown ~~as~~ in Figure 3.

```
Numerical source unit, [<i> NSU1,NSU2,NSU3,..NSUn]
header source unit, [hsu1,hsu2,hsu3,..hsun]
body source unit, [bsu1,bsu2,bsu3,..bsun]
Public accesss modifier weightage, P
Header value, [Hv1,Hv2,Hv3,..Hvn]
Body value, [Bv1,Bv2,Bv3,..Bvn]
Header ratio, [Hr1,Hr2,Hr3,..Hrn]
Body ratio, [Br1,Br2,Br3,..Brn]
Average header ratio, [AhR1,AhR2,AhR3,..AhRn]
Average body ratio, [AbR1,AbR2,AbR3,..AbRn]

1. Read a numerical source unit NSU1
2.   For header source unit hsu1
3.    Calculate Hr1 by dividing hsu1 with P
4.    Count code for header Hv1
5.    Calculate AhR1 by dividing Hr1 with Hv1
6.   For body source unit bsu1
7.    Calculate Br1 by dividing Bsu1 with P
8.    Count code for body Bv1
9.    Calculate AbR1 by dividing Br1 with Bv1
10. Repeat 1 to 9 on remaining available
    numerical source unit [NSU2,NSU3,..NSUn]
```

Figure 3. Pseudo code for Parameterization process.

EXPERIMENTAL RESULTS

This section will continue with the result of clones that have been found in two Python open source applications which are the Biopython and Natural Language Toolkit (NLTK) project. The discussion section is constructed within three subsections including the clone types definition, clone class, and clone pairs, and dataset limitation.

Clones in Biopython and NLTK

The result of clone pairs detected for the Biopython application is shown in Figure 4. There are 1667 clone pairs detected in Biopython application using the proposed solution. Among the 1667 clone pairs detected in the Biopython, 610 of the total clone pairs are detected as Type-1 and the remaining 1057 clone pairs are detected as Type-2. The proposed solution using Generic Code Clone Detection for Python language detects no clones with type 3 and type 4. Clone pairs with Type-2 holds the highest amount with 63.4% of the overall clones. While Type-1 is the lowest with only 36.6% of the total clones detected.

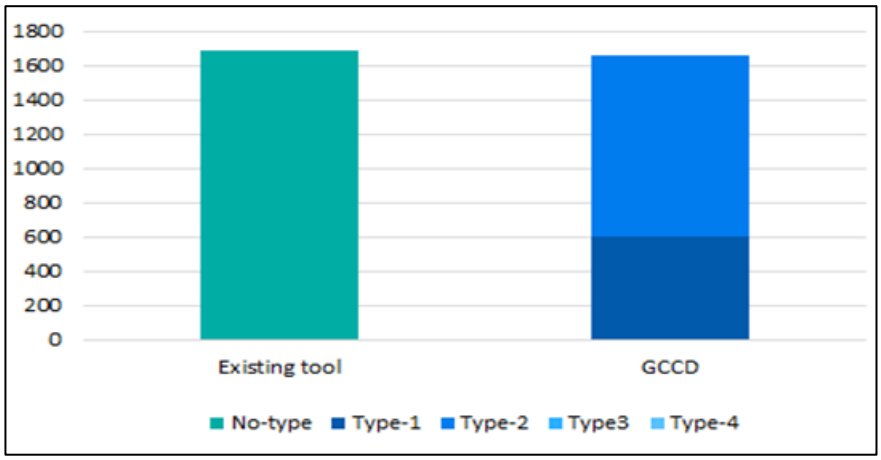

Figure 4. Clone Detection Result in Biopython.

The result of clone pairs detected for the NLTK Project application is shown in Figure 5. The proposed solution detected 277 total clone pairs in the NLTK application. Among the 277 clone pairs, 25 of the total clone pairs are detected as Type-1, 152 detected as Type-2, 73 detected as Type-3 and the remaining 27 detected as Type-4. The proposed solution using Generic Code Clone Detection for Python language detects Type-2 as the highest clone pair amount with 54.9% of the total clones. While Type-1 is the lowest with only 9% of the total clone pairs detected. From both Figure 4 and Figure

5, proposed solution using GCCD model for python application shows that Type-2 clones holds the highest amount of clone pairs in both Biopython and NLTK project application.

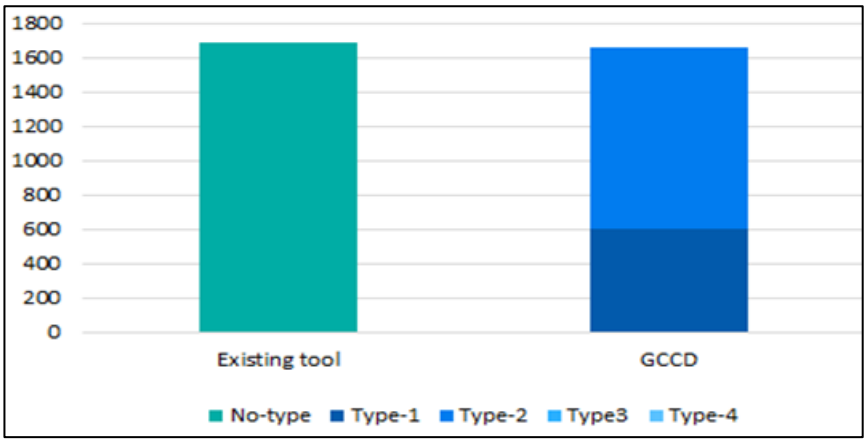

Figure 5. Clone Detection Result in NLTK

DISCUSSION

This proposed solution is done mainly to detects the existence of code clones in Python language applications and identify the clone types using the Generic code Clone Detection Model. The data collected within this work will be discussed in three areas. The first discussion will revolve around the definition of clone types, the second will be about detection approaches, and the third will be about the limitation of the dataset.

Clone Types

Code clone is also known as duplicated code [19][24] has a significant on the maintenance of software [20-23]. Different approaches and models might use a different definitions of clone types. Clone types are categorized according to the level of diversity and similarities of the clones. A different definition of clone types plays an important role as it highly affects the detection result. This work however is focusing on applying the common four types of clones that most researchers agree on [8]. The four common clone types are Type-1, Type-2, Type-3, and Type-4.

Type-1 is also known as an identical clone where both copied code and clone fragments are identical to each other with no alteration applied to the functional code except to comments and white spaces. Type-2 has slight changes where the parameter is renamed that make codes and clones are syntactically identical except for changes in name of variables, functions, and identifiers. Type-3 is where both copied code and clone fragments are modified in few parts including adding, removing, or changing few lines of code statements. Type-4 is known as a semantic clone where original codes and clones run similar functional computation but written in a different syntax.

Different Match Detection Approaches and Source Representation

Results can also be influenced by data representation and the match detection approaches used in the model and tools. The existing tool is representing their sources in the form of an abstract syntax tree (AST) [12] while this work however represents the source in numerical form. Different source representations will undergo a different processes. Source represents as a numerical form within this model is mainly to make the source measurable to produce matrices needed in the parameterization process.

The result for different clone detection models and tools also can vary due to different match detection approaches uses in the model. The existing tool uses the tree-based approach for the match detection process while this work is using Hybrid exact matching alongside Euclidean distance on the match detection process. Hybrid exact matching and Euclidean distance approach are crucial within the match detection process in this model to identify the clone types. Thus, different match detection approaches will produce different results.

Limitation of Dataset

This proposed solution aims to detect and identify code clones in Python applications. Application created using Python language can be considered hype in the tech world quite recently. Due to that, the existing data and findings revolve around clone detection in Python application to be compared with this GCCD for Python language is limited. Data collected from the only tool that is available to be compared with this work is also bounded due to different processes and approaches within both tools.

CONCLUSION

This enhanced solution in pre-processing process and parameterization process within the GCCD model is to detect and identify clones in Python language applications. The result of this work shows that it is possible for the GCCD model to detect and identify clones in Python language applications. Due to the limited findings in this field within python language, it is essential for this work to broaden the views of code clone existence in Python language.

ACKNOWLEDGEMENT

The authors would like to thank UMP for partially funding this work under an internal grant RDU190362.

REFERENCES

[1] B. Van Bladel, S. Demeyer, "Clone Detection in Test Code: An Empirical Evaluation.," in *SANER 2020 - Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering*, 2020.

[2] M. S. Rahman & C. K. Roy, "On the Relationships between Stability and Bug-Proneness of Code Clones: An Empirical Study," in *Proceedings - 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation, SCAM 2017*, 2017.

[3] M. Mondal, Is cloned code really stable?, vol. 23, Springer New York LLC, 2018, pp. 693-770.

[4] A. Sheneamer and J. Kalita, "A Survey of Software Clone Detection Techniques", International Journal of Computer Applications, vol. 137, pp. 1-21, 7 2016.

[5] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654-670, 7 2002.

[6] I. Baxter, "Clone detection using abstract syntax trees," *Conference on Software Maintenance*, pp. 368-377, 1998.

[7] B. Lague, D. Proulx, J. Mayrand and J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process", 1997 Proceedings International Conference on Software Maintenance, 1997, pp. 314-321, doi: 10.1109/ICSM.1997.624264.

[8] C. K. Roy, J. R. Cordy and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," Science of Computer Programming, Volume 74, Issue 7, Pages 470-495, 2009.

[9] Y. Yuan and Y. Guo, "Boreas: An accurate and scalable token-based approach to code clone detection," *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*, pp. 286-289, 2012.

[10] Y. Higo and S. Kusumoto. "Code clone detection on specialized PDGs with heuristics," *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp. 75-84, 2011.

[11] K. R. Srinath, "Python-The Fastest Growing Programming Language," *International Research Journal of Engineering and Technology (IRJET) 4.12 (2017): 354-357*, 2017.

[12] P. Bulychev and M. Minea, "Duplicate code detection using anti-unification" (Whitepaper), Retrieved from http://clonedigger.sourceforge.net/duplicate_code_detection_bulychev_minea.pdf. 2008.

[13] M. M. Morshed, M. A. Rahman and S. U. Ahmed, "A Literature Review of Code Clone Analysis to Improve Software Maintenance Process," *ArXiv*, vol. abs/1205.5615, 2012.

[14] C. K. Roy and R. C.James, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," 2007.

[15] M. Al-Fahim and S. Sulaiman, "Generic Code Clone Detection Model for Java Applications," *IOP Conference Series: Materials Science and Engineering*, vol. 769, no. 1, 6 2020.

[16] S. Giesecke, "Generic modelling of code clones BTC Energy Process Management Architecture (EPM) View project Generic modeling of code clones," Internationales Begegnungs- und Forschungszentrum f{\"u}r Informatik (IBFI), Schloss Dagstuhl, Germany , Dagstuhl Seminar Proceedings, 2007.

[17] B. Biegel and S. Diehl, "Highly configurable and extensible code clone detection," *Proceedings - Working Conference on Reverse Engineering, WCRE*, pp. 237-241, 2010.

[18] C. Kapser and M. W. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software. Empir Software Eng 13, 645 2008.

[19] Y. Golubev, V. Poletansky, N. Povarov and T. Bryksin, "Multi-threshold token-based code clone detection," 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021, pp. 496-500, doi: 10.1109/SANER50967.2021.00053.

[20] Y. Zhang and T. Wang, "CCEyes: An Effective Tool for Code Clone Detection on Large-Scale Open Source Repositories," 2021 IEEE International Conference on Information Communication and Software Engineering (ICICSE), 2021, pp. 61-70, doi: 10.1109/ICICSE52190.2021.9404141.

[21] V. Bandi, C. K. Roy and C. Gutwin, "Clone Swarm: A Cloud Based Code-Clone Analysis Tool," 2020 IEEE 14th International Workshop on Software Clones (IWSC), 2020, pp. 52-56, doi: 10.1109/IWSC50091.2020.9047642.

[22] M. J. I. Mostafa, "An Empirical Study on Clone Evolution by Analyzing Clone Lifetime," 2019 IEEE 13th International Workshop on Software Clones (IWSC), 2019, pp. 20-26, doi: 10.1109/IWSC.2019.8665850.

[23] M. S. Rahman and C. K. Roy, "A Change-Type Based Empirical Study on the Stability of Cloned Code," 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, 2014, pp. 31-40, doi: 10.1109/SCAM.2014.13.

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