Asia-Pacific Journal of Educational Management Research
Vol.5, No.3 (2020), pp.1-20
http://dx.doi.org/10.21742/ajemr.2020.5.3.01

Developing Machine Learning Coding Similarity Indicators for C and C++ Corpuses

Ajinkya Kunjir [1] and Jinan Fiaidhi[2]

[1]*Student, Department of Computer Science, Lakehead University, Thunder Bay, Ontario, Canada*
[2]*Professor, Department of Computer Science, Lakehead University, Thunder Bay, Ontario, Canada*
[1]*akujnir@lakeheadu.ca*, [2]*jfiaidhi@lakeheadu.ca*

Abstract

In the digital era of technology and advanced automation, data or information is vulnerable to copying, altering, and claiming someone else's work as their own. Source code theft or e-plagiarism is challenging to track in hundreds of assignments submitted by students. Despite the year's efforts, the digital plagiarism detection software currently available performs well enough for a naïve programmer to detect literal plagiarism. The available source code similarity detectors provide insufficient results when a student uses complex strategies such as word substitution or reordering programming constructs. To overcome the above-mentioned challenges, this research aims to deliver an assistive forensic engine for the professors and teaching assistants to evaluate the similarities in the student's assignments. This research's primary objective is to help the evaluators get closer to the sophisticated code thieves and abide by the university's academic dishonesty regulations. The proposed forensic similarity detection engine's constructive methodology is specially designed for studies where C and C++ programming languages are majorly used in academic assignments. After selecting the ATM (Attribute counting metrics), the system implementation is divided into two phases, where phase one consists of lexical analysis and tokenizer customization. The second phase mostly consists of rolling out the supervised learning algorithm on the generated data to classify the comparison of two files as a truth value. The similarity elements and observations recorded can be represented to the evaluators in the form of visualizations for ease of understanding and efficient decision making. The paper also relates the proposed system with the previous and existing system and mitigates the past issues noted in the latter half.

Keywords*: Similarity detection, E-Plagiarism, Tokenization, Lexical analysis, Distance algorithm, Euclidean distance*

1. Introduction

In a world full of advanced technology, searching for programming solutions, coding help, tutorials, and examples of source codes is a regular activity for programmers pursuing their topic of interest. Because of the high availability and convenience, digital documents can be

Article history:
Received (June 29, 2020), Review Result (September 4, 2020), Accepted (October 13, 2020)

Print ISSN: 2207-5380, eISSN: 2207-290X AJEMR
Copyright ⓒ 2020 Global Vision Press (GV Press)

easily copied, duplicated, and discarded across several platforms. The issues and challenges relating to digitalization have given rise to copying and cheating phenomena among individuals, which is often referred to as 'Plagiarism'. According to the most basic definition, plagiarism can be explained as copying/owning someone else's work without consent or not giving inventor credits to the initial owner. According to Parker et al. [1], a plagiarized program is a copied or modified version of another owner's source code with minor edit activities in the area of digital computer programming. In this research article, we will be focusing on e-plagiarism and similarity detection in C-C++ assignments submitted electronically to the web portal. Similarity detection is a complex mechanism for trivial plagiarism in source codes provided by undergrad and graduate students. The main objective of the proposed system is to aid/assist the teaching assistants to get closer to the source codes, which have a high degree of similarity. The detailed analysis and conclusions derived from the computation will be sufficient enough to target the plagiarized assignments submitted by the students. Plagiarism at an academic level is unacceptable as it brings no justice to the workers of original content. Every university now has an academic dishonesty regulation, which is an immoral and illegal act of plagiarism resulting in non-monetary penalties.

Source code duplication has been increased over the years and is problematic for the future of innovations. One among the first few researchers for plagiarism and similarity detection was Ottenstein [2] who published the first article in 1976 and emphasized more on operands and operators. Matija [3] described the never-ending work on source code similarity and plagiarism techniques and stated that the already existing tools had not been upgraded or updated for ten years, and also, there have not been any new inventions of relevance. Over the years, unskilled students' tendency has changed from adding more spaces and beautifying the code with comments to modifying the facial features such as identifiers, comments, an order of functions, and indentation of the code to avoid the same view or structural aesthetics. The sophisticated or skilled plagiarists are capable of altering the core components of code such as operators, declarations, expressions, constructors, control structures, and initializations. Al-Khanjari [4] in their publication on 'Plagdetect plugin' described the purpose of ATM's, SM's, and Hybrid techniques combining the formers ones. The authors made use of ATM (10 attributes in the matrix) combination with Equivalence ratio in their system to compute the similarities between multiple java class files. To shed some light on the sophisticated plagiarist tricks, consider the segment of the C++ code given below. The two sections belong to two student assignments, such as Student1 and Student2, wherein this case Student1 is the rightful owner of the code, and Student2 is a semi-skilled plagiarist. The second one has copied the former student's code and altered it in a decent way of not catching the eye of the examiner for a case of plagiarism or misconduct.

Table 1. C++ segment alteration: semi-skilled plagiarist

Student 1.cpp	*Student2.cpp*
#include <iostream> *using namespace std;* *int main() {* *int x, y;* *int sum;* *cout << "Type a number: ";* *cin >> x;* *cout << "Type another number: ";* *cin >> y;* *sum = x + y;* *cout << "Sum is: " << sum;* *return 0;* *}*	*#include<stdio.h>* *#include <iostream> //input output header file* *#include<string>* *using namespace std;* *int main() {* *int result; //variable to store the result* *int a, b;* *cout << "Enter a number ";* *cin >> a;* *cout << "Enter another number ";* *cin >> b;* *result = a + b;* *cout << "The result is: " << result;* *return 0;* *}*

It is very trivial for Student2 to modify the first assignment by simply putting fillers in the code and a few more edit operations. The changes made by Student2 in Student1's assignment are mentioned below as follows:

To get away with plagiarism on the first line, I added two more unneeded header files. (Changes highlighted in Student2.cpp section of the table)

The container variables for values have been renamed.. 'Int result' for 'int sum', 'x' and 'y' changed to 'a' and 'b'.

Marked two comments to introduce a new aspect if compared with assignment1.cpp

The above three edit operations allowed Student2.cpp to score 3-4 new lines in the code.

2. Literature survey

In the past few years, several research pieces have shown the statistics of plagiarism detection in a student programming environment in academics where marked assignment submission is involved. Faidhi et al. [5] in their research provided an in-depth analysis of program similarity and reported plagiarism for 'Pascal' programming language. The paper's literature survey consists of multiple software science measures included in the set of analyses, such as time complexity of the program, running time, length measures, absolute errors, and language level. A metric of 10 measures each was determined for program similarity. The first metric measures' m1 to m10' are supposedly intended towards a novice programmer's alterations, leading to an act of plagiarism. A few examples to give for the first set would be several characters, comments, indented lines, and blanks per line followed by the number of identifiers, reserved words, and variety of each. The second set of measures mostly attempts to qualify hidden/intrinsic features of program structure, which also indicates the flow of control. Talking about metrics and ratios, Al-Khanjari [4] addressed the plagiarism problem for beginner programmers in their critical review on 'PlagDetect', 2010. The novel research focused on evaluating the multiple existing metrics for plagiarism detection and then selecting an effective and appropriate ATM (Attribute counting metrics)

Copyright © 2020 Global Vision Press (GV Press)

for detecting similarities in java source codes. Unlike ATMs, SM techniques compare the program structures of multiple files and make up the result to spot the similarities.

Simply put, the SM approach begins with breaking down the source code into a stream of tokens and then compare the streams of programs to detect standard segments. Thomas McCabe [6] mentioned the complexity measure for the similar metrics described in the paper. The authors and inventors of the PlagDetect tool carried their final procedures with equivalence ratio (similarity coefficient) and ATMs for investigating java assignments, and validation has shown promising results in a comparative study executed against JPlag and YAP tools. The plagiarism detection method proposed in this paper is an effective combination of ATMS and SMs, also called a hybrid approach to define the combination of both techniques. The core idea is divided into two stages, such as initial lexical analysis and final comparison analysis after breaking down the code into tokens and lexemes further. The lexical tokenizer designed for lexical analysis is responsible for categorizing the lexemes into identifiers, keywords, mathematical operators, numerical operators, logical operators, and other operators. The other comment tokenizer alongside the lexical one is supposed to count the occurrences of comments with the line number and compare it with other submitted assignments in a parallel processing environment. In the second and the last phase of the proposed system, the string similarity distance between the segments of categories formed in the first stage is calculated using a plethora of distance algorithms in Java. The distance calculation is in chronological order starting from the first comparison such as *keywordi* = *keywordn*, math *operatorsi* = math *operatorsn*, and goes on till the last category. More details and information on the working system, comparison procedure, and summary of findings are explained in the later sections of this paper.

Zoran Djuric and Dragan Gasevic [7], in their research on the source code similarity detection system (SCSDS), describe the performance of their system when tested against the JPlag tool for detecting similarities when lexical and structural modifications are applied to the plagiarized code. The authors mention all kinds of lexical and structural changes a plagiarizer can edit in the paper to acquire in-depth knowledge about the study. In the comparative analysis with the JPlag tool, SCSDS outclassed the results on the test set with a huge difference between the statistical F-measures/scores of both the tools. The reason being the qualitative design of the tokenizer of the SCSDS tool, which holds meaningful semantic information about the lexemes. Adding more functionality to one system can improve the system's accuracy and reduce computing speed and worse time complexity. The authors from [7] added several similarity algorithms to increase the weights of the system dimensionality. The other limitation and disadvantages of SCSDS stated in the paper should be worked on in the future. Wise [8] from the department of computer science, University of Sydney, Australia illustrates the working of the first YAP tool's successor, YAP3 (Yet Another Parser). YAP3 is an improved version of YAP that focuses on an underlying algorithm: Running-Karp-Rabin Greedy-String-Tiling (RKS-GST). The specialty of the system is to detect transposed subsequences in the source codes. The system is not vulnerable to the source code manipulated with additional lines by a novice programmer. The system working for YAP3 is not complicated as the second phase after generating token sequences mainly consists of discarding comments & string constants, translating uppercase to lower-case, reordering functions, and passing the strings to the Karp-Rabin method parameters. According to the author of the paper [8], just like the Plague tool, even the latest version of YAP does not entirely parse the source code beyond identifiers and keywords. The ability of YAP3 to generate tokenizers provides flexibility to the weighted algorithm, and hence the functionality is expandable.

Whale [9] in his research mentioned that the popular previous approaches to plagiarism detection based on attribute counting measures are not adequate, and hence, the idea of the 'Plague' tool came into the picture. The mechanism of Plague works in a two-stage process where similar features are identified from the source codes in the population. The representation used for the final stage is the syntactic and semantic token analyzed from the first stage. The high selectivity potential is gained by detailed representation, whereas the high recall is achieved with the robustness of the least common subsequence's length calculations. The authors [10] who worked on solving semantic searches for source code aims to address the issue of complexity in the usage of semantic search approaches as the exiting approaches entail the programmer to define complicated queries to get desired results; however, the results often contain unnecessary matches that require manual filtration. The same authors have devised an approach where a programmer can use incomplete and lightweight specifications using an SMT solver that identifies programs from a repository that match the specifications provided by the programmer. The authors test the approach on subsets of the Java string library, Yahoo! Pipes mash-up language, and SQL select statements to gauge the effectiveness and efficiency that evaluates each domain. The results of the approach described in the paper are useful in the domains mentioned but the focus should be on the generality that addresses the problems of more complex constructs. The future scope of the approach is to test on a wider range of programs. The author in their paper [11] provides comprehensive information on plagiarism. A detailed account of plagiarism is, how it affects the research community, what accounts for plagiarism, and types of plagiarism provided by the authors. Along with that the method of plagiarism detection (manual and computer-aided) along with the process in which these methods work is discussed in detail. The paper's main focus is on the Free Text Detection technique under the Computer-Aided Technique that is categorized into the Lexical and Semantic-based detection technique. The authors provide comparisons of all the detection techniques and conclude that more work needs to be done in the Semantic-based detection technique. The paper also provides a survey on various offline and online tools available to detect plagiarism and an extensive comparison of various parameters.

3. Proposed system

As mentioned in the previous section, there are many complicated challenges to be faced with developing an automatic plagiarism detection system under certain situations. The first issue that needs to be addressed is setting up the corpus (set of documents) with the system's code design. The second challenge lead by the former one is the connectivity of corpus with the source codes for data retrieval and data processing by our plagiarism detector system. The proposed system is built using java programming language and trivial issues faced in the development process are class integration, database connection, front end setup, and data encapsulation. Figure-1given below explains the work-flow of the system starting from inputting the chosen dataset to ANTLR Tokenizer. To introduce ANTLR (Another tool for Language Recognition) in this paper is as a tokenizer for our programming language. This tokenizer consists of 'C' and 'C++' language grammar for processing the source codes and breaking them down into lexemes, also referred to as 'Lexical Analysis'. The lexemes are further processed through the 'Token Categorizer' where the lexemes are categorized into identifiers, keywords, mathematical operators, logical operators, and other operators. Following to categorization of lexemes, each group is fed to the distance similarity algorithms such as Jaro, Jaro-Winkler, Levenshtein, cosine similarity, dice coefficient, and least common

Copyright © 2020 Global Vision Press (GV Press)

substring for computation of values. Our similarity detection process can be explained in three stages given as under:

(1) Pre-Processing: This stage involves document(s) retrieval and uploads from the corpus set at the back-end. The document corpus consists of student assignments from the IEEE Homework programming dataset in this case, where each student 'N' submits a folder composed of 'Ni' number of assignments in it. Mathematically, a set of assignments' Ni' is a subset of student folder 'N' such as Ni ϵ N. In the proposed dataset, each subfolder 'Zi' ranging from 1 to 6 is within the subfolders A2016, A2017, B2016, and B2017. The details about the dataset are mentioned in section IV. Once the necessary documents are retrieved for comparison, they are fed to the file matching loop in the intermediate stage processing. The main objective of this pre-processing stage is to maximize the accuracy of document searching and to fetch from the corpus.

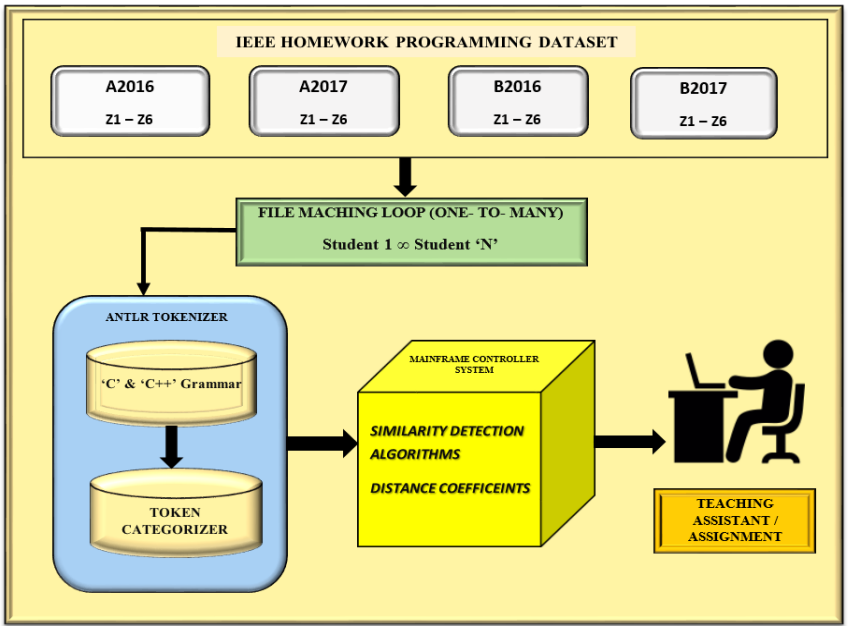

Figure 1. Hypothetical structure and work-flow of the proposed system

(2) Intermediate Processing: The second stage, followed by pre-processing, comprises three key components: File matching comparator or loop, ANTLR Tokenizer, and mainframe system component, which has similarity distance algorithms and controller functionality for the comparison. The detailed description of the working of each part mentioned above is given below as follows:

(a) File Matching Loop / Comparator - The 'N' student assignment folders are lined up in a working directory or path specified in the program. For a use case in undergrad school, all students are given multiple applications to code in the C/C++ programming language. Therefore, one assignment has three to four source codes at maximum in one folder when they electronically submit it to the subject professor.

The logic of comparing file 'A' from the first folder with all other files in the same folder makes sense without any extra loss of time. The comparator involved in the intermediate stage will compare 'Ni' of folder one with the rest 'n' numbers of 'Ni' going on till the last

file in the folder path specified. The comparator loop will continue matching the next file, i.e., Ni + 1, in line after 'Ni' from folder one and keep matching with the other files in the same folder. The files to compare are then passed to the sequence of tokenizers for breaking down the files into lexemes and teaming up with their categories such as keywords, mathematical operators, logical operators, numerical, and other operators.

(b) ANTLR Tokenizer- The general introduction of ANTLR is given at the start of section III. To dive deep into the tool, ANTLR uses the LL1 parser for reading and processing textual files. The plugin for ANTLR is available from its website (https://www.antlr.org/) and can be installed in the IDE environment such as an eclipse or IntelliJ IDEA. The ideal IDE platform preferred for developing this kind of system with heavy data handling and building grammars for parsing is IntelliJ IDEA. This tokenizer in the series is supposed to break down the stream of code into lexemes by referring to the 'C' or 'CC++' grammar. The program has been constructed in such a way that it can detect the extension of source codes in the given path such as '.c' or '.cpp' and choose the grammar file accordingly. The second tokenizer has exclusive use for detecting new lines, comments, and line numbers for the corresponding printouts. The lexical tokenizer, which is ANTLR, generates tokens in clusters of identifiers, keywords, arithmetic operators, logical and other operators for both the files and lists out the count for each cluster, including multiples. The next tokenizer in the queue has its expertise in detecting newlines and comments in the same set of files with the number of occurrences. The clusters/sets obtained from the source codes are compared with each other based on similarity distance algorithms in the mainframe system.

(c) Mainframe System- This essential component of the intermediate pre-processing stage has a collection of distance similarity algorithms to compare the clusters and give out the result. The ideal distance algorithms to be considered would be cosine similarity measure, N-grams, Levenshtein distance, Jaro, Jaro-Winkler, Sorensen dice coefficient, and least common substring the length of common subsequence. The performance evaluation and working of all the similarity algorithms based on distance methods are illustrated in the 'Similarity Measures Techniques' section in the latter half of this paper. The proposed research can also be framed as a comparative study of string similarity distance algorithms on a massive chunk of data where the results of all the categories of two files, i.e., keywords, math operators, logical operators, and other operators will be aggregated into one value for the ease of classification. The results obtained from this study will be summarized in a tabular format and displayed to the user operating at the front-end in the form of reports and visualizations.

(3) Post Processing: This is the final stage of the hypothetical structure of the system. The results from pre-processing and intermediate processing are validated in the form of reports and presented to the system's end-users. The stakeholders or users using the front-end of the system would be assignment evaluators such as Professors, Graduate assistants, and other fields related faculty. The result data/information could be visualized in charts, bar graphs, treemaps, column data charts, or maybe as simple as possible. The potential open-source tools considered and used in this research for representing analytics and visualizations are Google charts, ChartJS and Fusion Charts.

4. Data description

Generating source code datasets using artificial techniques is a challenging task and indirectly reflects various realistic situations. A good number of previous researches experimented on the output of 'jury' existing tool for measure algorithms [12]. As a lack of

Copyright © 2020 Global Vision Press (GV Press)

description of the standard datasets in existing researches, the new homework programming dataset is presented in this research to work with the proposed system. The 'Programming Homework Dataset for Plagiarism Detection' was uploaded on IEEE-Dataport by Vedran Ljubovic, University of Sarajevo [13]. The dataset is developed from the students' assignments for the subject – Introduction to C in one semester and assignments of C++ in other for the years 2016 and 2017. All the final source codes submitted by the students are available at http://ieee-dataport.org/open-access/programming-homework-dataset-plagiarism-detection and on AWS for comparison by the already existing plagiarism detection tools like JPlag, YAP3, MOSS, and PlagDetect. The homework assignment zip extract consists of full traces of student activity and keystrokes generated by setting the IDE to a time limit autosave during homework development. The IDE also helped passing out the output information from the compiler, debugger, and student assignment to a safe corner of the repository.

The instructions for the dataset go as an archive folder having three subparts in it as follows:

(1) Source codes – The actual source code assignments submitted by the students are stored in the /src folder inside the archive. The subfolders under 'src' are named A2016, A2017, B2016, and B2017. Each subfolder listed above contains more subfolders inside for individual assignments. Students were required to solve 16-22 assignments in each course, labeled as "Z1/Z1", "Z1/Z2", and "Z2/Z1" and so far till the end. The C/C++ source codes solved by the students are stored in these subfolders with an anonymous name. All the traces AutoSaved after every few seconds by the IDE are saved in the archive's stats folder. This folder is again segregated into subfolders named after courses, and the subfolder contains files ending with extension '.stats' for every student (name stays anonymous). The stats information is stored in JSON format (Key = value pairs). Figure 2 shown below gives a concept map view of the IEEE dataset where there are four courses- A2016, A2017, B2016, B2017, and assignments for each course is described as Z1/Z1.Z5/Z2 for each course.

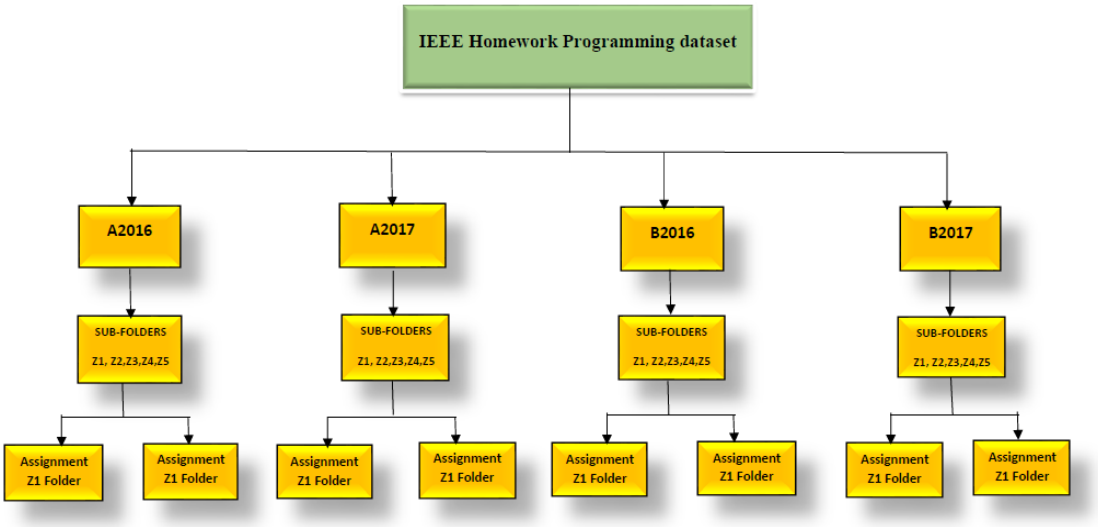

Figure 2. Dataset concept map

(2) Ground Truth - The instructions and format for JSON files is described in the readme.txt file present in the folder. The ground truth files list the individual and group of students involved in plagiarism due to code similarities detected in their assignments. The three ground truth files starting from 'ground-truth-anon.txt' contain a full list of plagiarisms, ground-truth-static-anon.txt based on source code similarity, and ground-truth-dynamic-anon.txt based on only failures due to 'oral defense'.

Some statistics generated by V. Ljubovic and E. Pajic[14] for the course ‘A’ in 2016 and 2017 i.e. A2016 & A2017 in their latest research published and accepted at IEEE in the year 2020 is shown below in [Table 2] as follows:

Table 2. Statistics for courses a2016 and a2017

Course	A2017
Student enrolled	488
Number of assignments	20
Submitted files	5733
Files per assignment	125-444
Average file size (bytes)	1317.23
Changes per file	1-7740
Plagiarized Solutions	699 (12.2%)

We all know that in a three-four-year-long course, the degree of homework participation, in the beginning, is way more than the involvement in the end. If the participation is 90% in the beginning, it closes up to 10-15% in the final semester of the course. As the willing participation increases, plagiarism decreases, and it’s vice versa in a long-term graduation program. The technique used to overcome the plagiarism index and balance out the proportion was to make 20% of the total students deliver oral-defense of their homework. The ground truth files were constructed on a marking system where the students who failed to defend their homework defense were marked as 'Plagiarized' in the file. Proper classification of homework is a must-needed feature in a similarity detector tool, but every tool handles the situation differently. Some tools have defined a threshold on assignment length. Some have pre-defined heuristics, and a few tools will simply mark all the students as plagiarized and leave unsupervised decisions to a human evaluator. A decent approach for avoiding overfitting with the proposed system in this paper would be to divide the dataset into training and testing datasets for the underlying machine learning algorithm. As explained at the beginning of this section, the normal ground truth file contains all the plagiarized files. In addition to the normal file, two more ground truth files have been added, such as static for similar homework documents and dynamic ones. They exclude original authors and keep those who have no similar pairs. In the ground file, the assignments are represented in similar files, such as triplets and quadruplets. When it comes to evaluating a newly developed plagiarism tool, one does not need to identify similar document pairs but should be able to count false positives and false negatives inclusive of detected pairs.

5. Similarity measure methods

According to the work-flow explained in the primary and intermediate stages of the proposed system, the source code breaks down into the number of lexemes/tokens and is forwarded to the tokenizers deployed within the lexical analysis phase. The tokens are mostly strings, integers, characters, and operators stored in separate containers or cluster sequences after the tokenizers categorize them into keywords, math operators, numeric operators, and

Copyright © 2020 Global Vision Press (GV Press)

others. The distinct sequences from containers in the file comparison stage are concatenated together in one sequence. They are evaluated with distance similarity algorithms to compute the similarity distance between the string sequences. In this section, we describe the potential distance-similarity algorithms taken into consideration for this research. Section VI gives a brief comparison of all the similarity algorithms with advantages, disadvantages, and limitations respectively.

5.1. Distance-similarity algorithms

(1) Levenshtein Distance

Levenshtein distance, also called edit distance, is defined as the similarity between two string sequences 1' and's 2'. The algorithm focuses on the minimum number of changes required to convert string's 1' into string's 2' with an operation such as insertion and deletion in string's 1'. In the programming area, the algorithm can be illustrated as lev (s1, s2) where the value lies between 0 and 1. The values closer to '0' indicate less similarity and nearer or equal to '1' indicate a greater measure of similarity. For example, lev (hello, hell) will fall somewhere between 0.8 and 1 as just one letter of's 1' is missing in's 2'. The mathematical equation for Levenshtein distance is given below as equation (1):

$$lev_{a,b}(i,j) = \int_0^{max(i,j)} min \int \begin{matrix} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{(ai\neq bj)} \end{matrix} \quad (1)$$

In the above equation, 1(ai≠bi) is the indicator function set to zero initially and equal to 1 otherwise. $Lev_{(a,b)}$ (i,j) = distance between first *i* characters of string 'a' and first *j* characters of string 'b'. The best example for Levenshtein distance between 'HONDA' and 'HYUNDAI' is 3 and edit changes using insertion, substitution, and deletion operations. The wider applications of Levenshtein distance in string matchings falls under dynamic programming, and the pseudocode for DP approach for Levenshtein distance is given below:

```
int LevenshteinDistance (char s[1..m], char t[1..n])
  // d is a table with m+1 rows and n+1 column
  declare int d [0..m, 0..n]

  for i from 0 to m
     d [i, 0]: = i
  for j from 0 to n
     d [0, j]: = j

  for i from 1 to m
     for j from 1 to n
     {
        if s[i] = t[j] then cost: = 0
                  else cost: = 1
        d [i, j]: = minimum (
                        d [i-1, j] + 1, // deletion
                        d [i, j-1] + 1, // insertion
                        d [i-1, j-1] + cost   // substitution
                     )
     }
  return d [m, n]
```

(2) Jaro Distance Algorithm

Like any other algorithm, Jaro similarity measures the distance between two string sequences. The value of Jaro (s1, s2) mostly ranges between 0 to 1, where two strings are equal when the value is 1 and not equal at all when a value is zero. The mathematical formula for Jaro and a detailed explanation on value calculation is given under as follows as equation 2:

$$Jaro\ Similarity = \begin{cases} 0, if\ x = 0 \\ \frac{1}{3}\left(\frac{x}{|s1|} + \frac{x}{|s2|} + \frac{x-t}{x}\right),\ for\ x! = 0 \end{cases} \quad (2)$$

From the equation above,

x = number of matching characters,

t = half the number of transpositions,

|s1| and |s2| = lengths of string s1 and s2

The matches are accurate if they are not farther than $\left[\frac{\max(|s1|,|s2|)}{2}\right] - 1$ and t = half the number of characters in both strings in a different order.

Consider s1 = 'rover' and s2 = 'flower', here the matching characters is three such as [o, e, r] in a different order. Number of characters not in order = 4 i.e. In s2 = [f, l, o, w]. Therefore, t = 4/2 = 2. From the above equation 2, Jaro similarity can be calculated as = 1/3 * ((3/5 + 3/6 + (3-2)/3) = 0.4719. The strings 'rover' and 'flower' have a Jaro similarity measure of 0 < 0.4719 < 1.

(3) Jaro-Winkler Algorithm

The Jaro-Winkler distance measure is similar to the Jaro algorithm in most cases where the prefix of both the strings doesn't match. They both produce different values when the prefix of both the strings doesn't match. The prefix scale 'p' in Winkler gives more accurate answers when strings have a common prefix of length 'L'. The Jaro-Winkler similarity measure is defined as follows in equation 3:

$$JW = J + Sf * L*(1 - J) \quad (3)$$

Where, J = Jaro measure obtained from above block,

Sf = scaling factor (0.1 by default),

L = length of matching prefix (max 4 characters long). Here for 'rover' and 'lower' we have L = 0.

The computation, JW = 0.4719 + 0.1 * 0 * (1 – 0.4719) = 0.4719. The Jaro-Winkler and Jaro in this case are equal. The results may be different for strings such as 'Logitech' and 'Lotto', where L =2.

(4) Sorensen Dice Coefficient (DC)

The Sorensen dice coefficient or dice index is a statistical tool used to gauge the similarity of two samples. This invention intended to differentiate the similarity between two distinct sequences. Assume '1' and '2' to be two distinct data sequences and |s1| and |s2| be the cardinalities of the same sets. The dice index /coefficient equals twice the number of elements common to both sets divided by the sum of cardinality sets. The mathematical equation for DC or DI (Dice Index) is given below in equation 4 as follows:

$$DCS = \frac{2\ |s1 \cap s2|}{|s1| + |s2|} \quad (4)$$

The only difference between the Jaccard coefficient and DCS is that Jaccard counts the true positives once in both denominator and numerator and DCS falls in between 0 and 1 for

Copyright © 2020 Global Vision Press (GV Press)

two discrete sets. The DCS for string similarities is a variance of the normal DCS form and uses bigrams of the strings for computation, as shown in equation 5:

$$DCS\ for\ Strings = \frac{2nb}{nx+ny} \quad (5)$$

Here, 'nb' is the number of bigrams found in both strings, and 'nx' & 'ny' denote the number of bigrams found in string X and Y, respectively. Consider the words' Deer' and 'Dear', the set of bigrams in each word world be as follows:
X = {de, ee, er}, Y = {de, er, ar}

The common bigram between both the strings is {de}. Therefore, the DCS we obtain after calculation by putting in equation (5) is (2.1) / (3 + 3) = 0.33.

(5) Longest Common Substring (LCS)

Another Dynamic Programming (DP) approach considered in this proposed similarity detection engine after Levenshtein distance is the longest common substring (LCS) for assuring string similarity without any resulting numerical value. Being a DP implementation, this algorithm has a time complexity of O(nm) where space is utilized more than time. The definition of LCS is simple as it identifies a substring in's 1' and checks for the same in's 2'. The algorithm also has the functionality of keeping track of the substring's maximum length and displaying it on the console. An example for LCS detection is given below in words as there is no exclusive statistical explanation for it in the world of algorithms.

k-common substring problem ϵ LCS *(X, Y, m, n)* = Max (LCSuff(X,Y,I,j)) where 1 <= i ,j <= m, n. Max (LCSuff) is the equation where both the strings lengths is reduced by 1 if the last characters match.

Local similarity algorithms

(1) Cosine Similarity Measure

Cosine similarity can be defined as a document similarity metric that is used to measure the similarities between two documents irrespective of the size. It measures the cosine of the angle between two vectors in a 2D multi-dimensional space. The vectors selected for measurement can be strings, arrays, and value objects in a coded algorithm. The core programming language used for developing the proposed system is Java, and hence, forming the vectors from the tokenization approach is not cumbersome in the procedure. The main advantage of this method is it can conclude that two documents can be oriented together even if they're far apart because of size irregularities. Like the other techniques stated above, the result value of cosine similarity ranges between 0 and 1. The similarity percentage is less if the cosine angle is big and high when the angle is small. The cosine similarity is implemented for document similarity in two ways as described below:

Approach 1: Consider 'A' and 'B' as two document vectors and measure the cosine similarity angle between the two vectors to justify the similarity between two documents in the range of 0 to 1. This approach is favorable for the researchers, which focuses on occurrences of a word for checking document similarity.

Approach 2: Tokenize the document to form categories for simplification and then concatenate the distinct features into one complete vector. Follow this procedure for all the documents and then calculate the cosine angle between the vectors. The result for this approach would be more effective than approach one as the vector would contain all distinct elements from all the categories. The mathematical formula for cosine similarity is given under equation 6:

$$Cosine\ Similarity = cos(\theta) = \frac{X.Y}{||X||||Y||} = \frac{\sum_{i=1}^{n} X_i Y_i}{\sqrt{\sum_{i=1}^{n} X_i^2}}. \frac{1}{\sqrt{\sum_{i=1}^{n} Y_i^2}} \quad (6)$$

In the equation above, 'X' and 'Y' are the two vectors of attributes, and cosine similarity is represented as a dot product and magnitude. The result obtained from this formula will be '1' if the documents are clones and '0' if they're the opposite. In the case of IR (Information retrieval) the angle between two '*term*' vectors cannot be > 90 degrees. Gunawan [15] in their research on finding text relevance via cosine similarity mentioned the use of cosine similarity measures to find the relevancy of a suitable topic in multiple documents. The authors divided the system implementation into three stages such as pre-processing (removing punctuations from documents, converting all text to lower-case, etc.), intermediate (keyword weighing between 0 and 1) and the last stage involves cosine angle measure to give out relevancy in terms of '0' or '1'.

(2) N-grams similarity measure

The core concept and motive behind n-gram similarity is improvising the concept of the mainstream LCS (Longest Common Subsequence) to encompass n-grams rather than unigrams. Assume n-gram similarity as function 'Sn', where 'n' is a fixed parameter of 'S'. The other widely used measure other than LCSR (LCS Ratio = LCS / length of the longer string) is NED (Normalized edit distance) for n-grams, n>1. We need to emphasize more on normalization and affixing before moving forward with n-gram for word similarity. Normalization is defined as a method of discounting the words being compared. The length of LCS of generated strings grows in-hands with the length of the strings. For Similarity X Distance, when n =1, the similarity is LCSR, and the distance will be NED. For n =2, the similarity is BI-SIM and distance as BI-DIST. The classification measure goes on as Tri, quad to 'n' where n = [3, 4….' n']. Affixing adds sensitivity to the symbols at string boundaries, which qualifies them for participation in fewer n-grams than internal symbols. The authors in the paper [16] proposed n-gram similarity and distance measures N-SIM & N-DIST incorporate normalization and affixing to enhance the n-gram procedure's performance. The simple definition of n-gram goes as 'common sequence of 'n' items from a sequence of speech and the things that can be letters, characters, or words. Saima Sultana and Ismail Biskri [17] in their research described a methodology that uses the notion of n-gram characters due to their nature of the sentences to be analyzed. The primary step of the methodology is to assume two short sentences such as 'Similarity' and 'Dissimilarity' and remove the special characters, punctuations, and stop-words.

Convert all the letters and characters to lower-case as 'similarity' and 'dissimilarity'.

Producing n-grams of characters of the strings where n = 1, n = 2 (Bigrams) and n = 3 (Trigrams) etc. till n = ‘n’.

Using trigrams, the example will be:

String 1 = similarity = ‘sim’,’imi’,’mil’,’ila’,’lar’,’ari’,’rit’,’ity’

String 2 = dissimilarity = ‘dis’,’iss’,’ssi’,’sim’,’imi’,’mil’,’ila’,’lar’,’ari’,’rit’,’ity’

Forming two distance matrices to calculate the distance between the pair of 'n-gram' characters leads to measuring the similarity between two large strings. For example, (sim, sim) from (string1, string2) will be '0' and the same for (imi, imi), (ity, ity), (lar, lar), and so on. The positive and negative correlation values can also be obtained and referred to as distance.

The next step would be to remove negative values as the distance cannot be negative.

Copyright © 2020 Global Vision Press (GV Press)

Calculate the similarity and dissimilarity from the two matrices for the given two strings. The precise values can be obtained by using popular co-occurrence measures such as dice, overlap, Jaccard and cosine.

(3) Greedy String Tilling – Karp Robin Problem (GST-KRP)

Michael J. Wise [18] proposed the use of the "Greedy String Tiling and Running Karp-Rabin matching (GST - KRP)" algorithm in the areas of plagiarism/similarity detection and other applications such as DNA Matching, amino acids, etc. GST algorithm is based on the concept of one-to-one string matching and can deal with the substring transposition. A Running Karp-Rabin method, which has an average close-to linear-complexity has been suggested for computing GST values wisely. The three important features of the GST algorithm are 'maximal match', 'tile', and 'minimum match length'. We will describe the working of the algorithm along with the definition for each of the above-mentioned features in this section as given below.

Let 'P' be the pattern (Pattern is usually the shorter string among the two strings) and T be the text string.

Maximal-Match: In any line of comparison with GST, a match is said to be a 'maximal match' when the substring P_p of the pattern string 'P' starting at p matches element-by-element, a substring T_t of text T (the other string). The match goes on until the end of the string is reached. A maximal match can be shown as max (p, t, s), where s = length of the match, 'p' & 't' are matching points.

Tile: A tile is a one-to-one association of substring of P with a matching substring of T. For tile formation from a maximal match, tokens of two substrings are marked and become unavailable for further matches. In simple words, tile formation is directly dependent on a maximal match and can be denoted as tile (p, t, s) where s is the length of the tile.

Minimal Match: A minimal match is just a maximal match below some defined length threshold. A minimum match length can be one but a value > 1. Assume that there is a current maximal match length 'maxmatch' (maxmatch >minmatch) which is the length of the largest maximal-matches obtainable from P and T. The algorithm =, more or less, the pseudocode: provided by the author from the paper [19] has been provided in the code box given above. To optimize the regular GST algorithm, the minimum match-length was put greater than 1, and the results obtained have the optimal values. The worst-case - time complexity of GST being O(n3), the algorithm can be tuned in several ways to improve the performance and reduce the worst-case complexity. The explanation of Karp-Rabin or Running-Karp-Rabin and how the novel algorithm computes GST values will be explained in the second version of this paper.

```
Greedy-String-Tilling Algorithm
Tiled_tokens_length: = 0
Repeat
maxmatch: = minimum-match-length
starting at the first unmarked token of P, for each Pp do
starting at the first unmarked token of T, for each Tt do
j: = 0
while Pp+ j = Tt+ j AND unmarked (Pp+ j) AND unmarked (Tt+ j) do
j: = j + 1
if j = maxmatch then add match (p, t, j) to list of matches of length j
else if j > maxmatch then start new list with match (p, t, j) and maxmatch: = j
for each match (p, t, maxmatch) in list
if not occluded then /* Create new tile */
for j: = 0 to maxmatch - 1 do
mar k_token (Pp+ j)
mar k_token (Tt+ j)
Tiled_tokens_length: = Tiled_tokens_length + maxmatch;
Until maxmatch = minimum-match-length
```

6. Comparative study of existing tools

Over the past few years, automated similarity detection engines have been developed for identifying plagiarism in student source codes. The previous researches have stated that the most effective approaches have been about tokenizing student assignments into bits and pieces and looking for long common substrings/subsequences between the sequences. The researchers seem to follow structured metrics referred to as the development of MOSS, PlagDetect, JPlag, and YAP tools. The attribute counting metrics (ATMs) and similar measurements designed for the approach consist of various measures already mentioned in section I – Introduction of this paper. The early plagiarism detectors tools such as JPlag, Sherlock [20][21], Sim, and Plaggie have been discussed quite a few times in the research papers. 'Sim', 'Plaggie' and 'JPlag' are software that uses a token-based approach for detecting similarities in programming assignments, and 'Sherlock' is a tool based on digital signature recognition for plagiarism checking. There are several other e-plagiarism checkers available for study but could not be verified practically as the command-line version was never developed for them and could not be automated for public use. To pursue a case study of all available tools, quantitative measurement is required to generate a value 'i' for comparing (x, y), where 'x' and 'y' are two files in a basic comparator. Chaiyong Ragkhitwetsagul [22], in their research paper on the comparison of code similarity analyzers, described the five experimental scenarios for pervasive modifications: obfuscators, clone detectors, and other software in regards to all the available similarity detectors in or before the year 2017. Fetching some information from the referenced paper, the table given below shows the list of tools with their similarity measures, details, default parameters, and year of development. Table 3 provides the audience with a detailed comparison of all categories of similarity detectors such as plagiarism detection tools (PD), clone detector tools (CD), and others (O), which also includes compressors and mini-tools. In addition to the comparison of the tools with their similarity measurement calculation, we have added details, default parameters, and the year of the invention along with the research paper reference in the columns of the same table.

Copyright © 2020 Global Vision Press (GV Press)

Table 3. A comparison table showing similarity calculation, details, default parameters, year, and references for each tool

Tool category	Similarity Calculation	Details	Year and Reference
(PD) SIMTEXT	Tokens and string alignment	Min. run size	1999 Gitchell and Tran [18]
(PD) SIMJAVA	Tokens and string alignment	Min. run size	1999 Gitchell and Tran [18]
(PD) SHERLOCK	Digital signatures	Chain length, zero bits	2002 Pike R and Loki [19]
(PD) JPLAG-TEXT	Tokens, GST(Greedy String Tiling), Karp-Rabin	Min. no. of tokens	2002 Prechelt [20]
(PD) JPLAG-JAVA	Tokens, GST(Greedy String Tiling), Karp-Rabin	Min. no. of tokens	2002 Prechelt [20]
(CD) CCFX	Tokens and suffix tree matching	Min. no. of tokens	2002 Kamiya [21]
(CD) YAP	Tokens, GST(Greedy String Tiling), Karp-Rabin	Tokenization and GST matching	1996 Michael J. Wise [8]
(PD) PLAGGIE	N/A (Not disclosed)	Min. no. of tokens	2006 Ahtanein [22]
(CD) DECKARD	Characteristic vectors of AST optimized by LSH	Min. no. of tokens Sliding window size Clone similarity	2007 Jiang [23]
(CD) NICAD	TXL and string matching (LCS)	Percentage of unique code Min. no. of lines Code abstraction Variable renaming	2008 Roy and Cardy [24]
(CD) ICLONES	Tokens and generalized suffix tree	Min. of tokens	2009 Gode and Koschke [25]
(O) COSINE	Cosine similarity from machine learning library	N/A	2011 Pedregosa et al [26]
(O) FUZZYWUZZY	Fuzzy string matching	Similarity calculation	2011 Cohen [27]
(O) NGRAM	Fuzzy search using n-gramme	N/A	2012 Poulter [28]
(CD) SIMIAN	Line-based string comparison	Min. no. of lines Ignoring variables, whitespaces, identifiers	2015 Harris [29]
(O) DIFFLIB	Gestalt pattern matching	Ignoring whitespace, auto junk heuristic	2016 Python Software Foundation [30]
(O) DIFF	Equation	N/A	2016
(O) BSDIFF	Equation	N/A	2017
(O) JELLYFISH	Approximate and Phonetic String matching	Edit distance algorithm	2016 Turk and Stephens [31]
(C) 7ZNCD	NCD with 7z	Compression level	N/A
(C) BZIP2NCD	NCD with bzip2	Compression level	N/A
(C) GZIPNCD	NCD with gzip	Compression level	N/A
(C) XZ-NCD	NCD with xz	Compression level	N/A
(C) ICD	Regular NCD (Normalized Compression Distance)	Compression level, block size	N/A
(C) NCD	Regular NCD	Compression level	2015, Cilibrasi [32]

The authors also validated the performance of all the tools mentioned in ［Table 3] above with a dataset consisting of java source codes mentioned in the paper [33][34][35][36]. The performance factors evaluated were Truth values (T), false positive (FP), false negative (FN), accuracy, precision, recall, the area under curve (AUC), and F1-score. The detailed comparison of the proposed forensic engine's previous tools based on the similarity measures/calculations will be discussed in the successor of this current paper.

7. Conclusions

In many academic institutions, source code plagiarism is still an ongoing concern and disrespects academic awards' moral integrity. Several students digitally submit their assignments to the repository and this makes it challenging for the evaluator to check and compare one assignment with others for plagiarism. The existing similarity detection tools use inefficient approaches such as Attribute Counting Metrics (ATM) with the tokenization approach that involves the Longest Common Substring (LCS) search method. A bunch of similarity detector engines prefers using hashing techniques and syntax tree/AST modifiers for file matching if the focus is on the line-word comparison. It is a complicated decision to make when it comes to recommending a tool above all others. MOSS, YAP3, and JPlag are well used within the professional academic community because of their various advantages for all kinds of programming language. Few notable disadvantages of these similarity detectors are lack of visual support (GUI), batch file processing, and a robust assistant tool. The similarity detection engine proposed in this paper tries to resolve the complexities and challenges faced by the evaluators and examiners at professional institutes where students upload their assignments digitally. The system follows a systematic ATM alongside a tokenizer (ANTLR) driven mainframe controlling system delivering lexical analysis computation with multiple algorithms. The IEEE homework programming dataset comprising of 'c' and 'cpp' courses assignments is given as a path to the program and assignments are evaluated in batches. The similarity measures considered for this experimentation include cosine similarity, n-grams, Levenshtein distance, Jaro & Jaro-Winkler, and coefficients such as Dice, Jaccard, and F-1. An average score of all these methods is obtained to classify if two assignments are plagiarized or not. Adding a novelty feature to this implementation apart from the detection process, the research intends on developing a web application for representing analysis of student assignment comparison and a machine learning touch for classification of a contrast.

8. Future work

The current research can be expanded in the future by extending the detection process to the next level, which is syntactical analysis. The construction of a parser tree using ANTLR for one source code is complex and therefore will be more difficult to do the same for a bunch of files in a parallel processing environment. The expansion will improve the comparison accuracy as the source code controls, and constructs will be evaluated. Various parse tree algorithms for recursive descent parser and LR/LL can be used for similarity detection. Diagrammatic representations of critical analysis and insights within the comparison process will be essential and play a key role in the future for this kind of research.

Acknowledgment

This paper is part of the first author MSc Thesis.

Copyright © 2020 Global Vision Press (GV Press)

References

[1] A. Parker and J. Hamblen, “Computer algorithms for plagiarism detection,” IEEE Transactions on Education, vol.32, no.2, pp.94-99, May

[2] K. J. Ottenstein, “An algorithmic approach to the detection and prevention of plagiarism,” SIGCSE Bull, vol.8, no.4, pp.30-41, December

[3] M. Novak, M. Joy, and D. Kermek, “Source-code similarity detection tools used in Academia: A systematic review,” ACM **(2019)**

[4] Z. Al-Khanjari, J. A. Fiaidhi, R. Al-Hinai, and N. S. Kutti, “PlagDetect: A java programming plagiarism detection,” Plug-in. ACM Inroads magazine, **(2010)**

[5] J. A. W. Faidhi and S. K. Robinson, “An empirical approach for detecting program similarity within a university programming environment,” Computers and Education, vol.11, no.1, pp.11-19

[6] T. McCabe, “A complexity measure,” IEEE Transactions on Software Engineering, vol.2, no.4, pp.308-320

[7] Z. Đurić and D. Gasevic, “A source code similarity system for plagiarism detection,” The Computer Journal, vol.56, pp.70-86, **(2013)** DOI: 10.1093/comjnl/bxs018

[8] M. J. Wise, “YAP3: Improved detection of similarities in computer program and other texts,” ACM SIGCSE Bulletin, vol.28

[9] G. Whale, “Identification of program similarity in large populations,” The Computer Journal, vol.33, no.2, pp.140-146

[10] K. T. Stolee, S. G. Elbaum, and D. Dobos, “Solving semantic searches for source code,” **(2012)**

[11] S. Yousf, M. Ahmad, and S. Nasrullah, “A review of plagiarism detection based on lexical and semantic approach,” pp.1-5, **(2013)** DOI:10.1109/C2SPCA.2013.6749430

[12] M. Mozgovoy, “Enhancing computer-aided plagiarism detection,” Ph.D. Dissertation, University of Joensuu, Department of Computer Science and Statistics, **(2007)**

[13] V. Ljubovic, “Programming homework dataset for plagiarism detection,” IEEE Dataport, Aug., **(2020)** DOI: 10.21227/71fw-ss32

[14] V. Ljubovic and E. Pajic, “Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories,” in IEEE Access, vol.8, pp.96505-96514, **(2020)** DOI: 10.1109/ACCESS.2020.2996146

[15] R. Cilibrasi and P. M. B. Vitanyi, “Clustering by compression,” Trans Inf Theory, vol.51, no.4, pp.1523-1545, **(2005)**

[16] D. Gunawan, C. A. Sembiring, M. A. Budiman, “The implementation of cosine similarity to calculate text relevance between two documents,” Journal of Physics: Conference Series. 978. 012120. 10.1088/1742-6596/978/1/012120, **(2018)**

[17] G. Kondrak, “N-gram similarity and distance,” in: Consens M., Navarro G. (eds) string processing and information retrieval, SPIRE 2005, Lecture Notes in Computer Science, vol.3772, Springer, Berlin, Heidelberg, **(2005)** DOI: 10.1007/11575832_13

[18] S. Sultana, and I. Biskri, “Identifying similar sentences by using n-grams of characters,” **(2018)**

[19] J. Turk and M. Stephens, “A python library for doing approximate and phonetic matching of strings,” https://github.com/jamesturk/jellyfish, **(2016)**

[20] M. Joy and M. Luck, “Plagiarism in programming assignments,” Technical Report, University of Warwick, Coventry [PDF] (BibTeX)

[21] M. S. Joy and M. Luck, “Plagiarism in programming assignments,” IEEE Transactions on Education, vol.42, no.2, pp.129-133

[22] C. Ragkhitwetsagul, J. Krinke, and D. Clark, “A comparison of code similarity analyzers,” Empire Software Eng, vol.23, pp.2464-2519, **(2018)** DOI: 10.1007/s10664-017-9564-7

[23] G. David and T. Nicholas, "Sim: A utility for detecting similarity in computer programs," SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), vol.31, pp.266-270

[24] R. Pike and Loki, "The sherlock plagiarism detector," https://www.sydney.edu.au/engineering/

[25] L. Prechelt, G. Malpohl, and M. Philippsen "Finding plagiarisms among a set of programs with JPlag," Journal of Universal Computer Science, vol.8, **(2003)**

[26] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFInder: Amultilinguistic token-based code clone detection system for large scale source code," Trans Softw Eng, vol.28, no.7, pp.654-670, **(2002)**

[27] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: GNU-licensed source code plagiarism detection engine for java exercises," In: Baltic sea, no.6, pp.141-142, **(2006)**

[28] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," pp.96-105, **(2007)** DOI: 10.1109/ICSE.2007.30

[29] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty printing and code normalization," In: ICPC'08, pp 172-181, **(2008)**

[30] N. Gode and R. Koschke, "Incremental clone detection," In: CSMR'09, pp 219-228, **(2009)**

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, and V. Dubourg, "Scikit-learn: Machine learning in python," J Mach Learn Res, pp.2825-2830, Oct,. **(2011)**

[32] A. Cohen, "Fuzzywuzzy: Fuzzy string matching in python," http://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/, accessed date: 14 Mar., **(2016)**

[33] G. Poulter, "Python n-gram 3.3," https://pythonhosted.org/ngram/, accessed date: 14 Feb., **(2016)**

[34] S. Harris, Simian - similarity analyzer, version 2.4. http://www.harukizaemon.com/simian/, accessed date: 14 Feb., **(2016)**

[35] Python Software Foundation Difflib – helpers for computing deltas, http://docs.python.org/2/library/difflib.html, accessed date: 14 Feb., **(2016)**

[36] M. Wise, "String similarity via greedy string tiling and running Karp−Rabin matching," Unpublished Basser Department of Computer Science Report

Copyright © 2020 Global Vision Press (GV Press)

This page is empty by intention.