Qualitative Estimation of Plagiarism Presence in Programming Assignment Submissions

Tetiana Karnaukh

Taras Shevchenko National University of Kyiv, 64/13, Volodymyrska Street, Kyiv, 01601, Ukraine

Abstract

The aim of this paper is to develop an approach for qualitative estimation of plagiarism presence in programming assignment submissions. Desired algorithm should take into account that some parts of assignment can be discussed in class, can implement design patterns or can contain pieces of code provided by instructors. The distinctive feature of processed data is that analyzed submitted programs solve similar tasks and therefore cannot be very different. So, we are trying to find the most similar programs between similar ones.

An approach to qualitative estimation of plagiarism presence among the set of programs for similar tasks is formulated. According to it, all the submissions should be processed simultaneously. To analyze a submission, a multiset formed by the numbers of n-grams common with other submissions is computed. The decision about submission originality, i.e., plagiarism absence, is based upon the density of this multiset largest elements.

The proposed technique is simple to implement for any programming language course and is quite effective to help an instructor to recognize signs of borrowings in programming assignments. It is used by the author since 2018.

Keywords [1]

Plagiarism, programming assignment, similarity, n-grams, density

1. Introduction

The problem of text document similarity estimation has large applied importance today. Due to a large amount of information, people need to automate the processes of distinguishing unique elements as well as the dual processes of finding out similarities. As a result, numerous methods of similarity estimation are used in the searching systems and so on. One branch of such methods is focused on the analysis of texts on plagiarism. Among all texts, one can mark out program texts as texts with some specifics. And, finally, estimation of arbitrary programs similarity is not a goal of this paper. More precisely, this paper is concerned about how given a set of quite similar programs, one can find out the most similar ones. The input data consist of all the submissions of some programming assignment. A key point is that this assignment can be only slightly different for different students. For each program from the set, the problem is to decide if a given program is original or it borrows some essential parts from another program that we have.

In general, the methods of similarity estimation are well known. But among them, there are no universal method that would give an adequate similarity estimation in all the domains.

A large family of methods for texts similarity evaluation is based upon n-grams (in [1] are mentioned as n-shingles) technique and Jaccard similarity of sets [1]. Different n-grams techniques are considered in [1-4]. Another group is based upon finding common subsequences, for example [5]. Methods of both groups can do text preprocessing. For natural languages it can be, for example, lemmatization. Considering the specifics of program code, methods for code analyzing transform code into a sequence of tokens, which are not necessary tokens of corresponding programming language, but semantically quite close to them [5, 6], and then analyze induced sequence. There are methods using abstract syntax

Information Technology and Implementation (IT&I-2022), November 30 - December 02, 2022, Kyiv, Ukraine
EMAIL: tkarnaukh@knu.ua (A. 1)
ORCID: 0000-0001-6556-1288 (A. 1)
© 2022 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3384/Short_9.pdf

tree [7] and programs behavior [8-10]. Programs similarity is interconnected with different plagiarism attacks [11] or different ways of code cloning [12, 13]. One of the modern tendencies is applying methods of AI to problem of program similarities [13, 14], but these techniques use significant computing resources and need a lot of data for training.

In methods [5-8] after lexical analysis, when a sequence of programming language tokens is derived, one should make additional transformation, and this transformation depends on the programming language. For modern popular programming languages, there are tools for lexical analysis so that it is possible to use existing libraries to transform code into programming language token sequence. But additional transformation should consider syntactic as well as semantic programming language specifics. Thus, such transformation should depend on programming language too. In case of evolutive programming languages, for example Python, making frequent update of this transformation for a more modern language dialect can be quite inconvenient. Sometimes language changes can invoke minor changes in the additional transformation. But sometimes one can think of more significant changes which could not be easily worked out and implemented.

It is worth of pointing out that methods which simply estimate similarity of two programs are not very applicable to the situation in which it is necessary to define the most identical objects among the moderate amount of almost identical ones. In such circumstances, one should consider that the similarity of submissions with no plagiarism can be quite large.

2. Strategy of programming assignments composing and usage of proposed technique

Each programming course is accompanied with series of programming assignments for unsupervised work. These assignments can be very simple tasks for learning certain programming techniques and constructions as well as more complicated programming projects for acquiring skills not only in coding, but in planning work, designing a project, testing and so on.

Modern programming is built upon patterns and libraries. Following the notion of good code, programmers use meaningful names, which for typical programming entities are quite typical. As usual, the best programs for very easy assignment do not vary too much due to using well-known patterns, and this is normal. Overachiever students write almost identical version of, for example, matrix multiplication. If such assignment is given to 100 students, their programs can be grouped into some not very large number of clusters. Moreover, for a very simple program we often could not assert that it was done with collaboration with someone else even in case of 90% program text coincidence. In most cases such simple assignments are not supposed to be graded. For more complicated assignment, a student has more alternatives. He or she should decide which patterns to use and how to implement these patterns. Also, a student should make other decisions about the code and its structure. As a result, such programs can include not only implementations of standard design patterns but some original parts. Their code can be near about 10 kB (the author deals with first-year students). A great problem with such assignments is that some students tend not to program them on his or her own. Sometimes students understand borrowed code but sometimes not, and the last happens much more often. An instructor should be able to identify not written by one's own programs because such assignments should not be graded positively. One can think that if we assign each student a substantially different task, then we can eliminate the problem of code plagiarism. From one side, this is true; but from the other side, this is not a solution. First, it is a problem for instructors to invent 100+ different but equivalent in complexity tasks each year. Then, the programming course includes some classes where the lab instructor and students discuss some design patterns and the instructor guides students through their tasks gradually. In case of very distinct assignments, there can be no common guides suitable to all tasks simultaneously. Finally, if tasks are very different and there are no guides, then students will begin to find external help with more probability. Our teaching goal is to stimulate students to do the work by themselves and to teach them to solve not only the simplest problems so that the above-mentioned approach is not suitable. An opposite approach is to give each student the same assignment. This is not a good idea too because provokes students to borrow code from each other.

A possible solution is to use similar, but not the same tasks. This let instructors not only guide students through their tasks but let them automate testing (to be more precise, automate test generation because automation of testing is not a problem if we have tests). Different in details tasks make students deep into the samples and guides because they cannot be used literally. Unfortunately, some students

prefer to ask other students for code anyway. It does not mean that one student copies the whole program of other. He or she can copy some parts, also he or she can slightly modify borrowed code and he or she can have quite different set of the translation units. Borrowed parts do not always coincide literally. From the other side, students never write the same code after discussions in the class. Their programs are more different than in case of simple copy-paste method.

Instructions, guides, requirements to code, and other like that have great influence on the design and code of the resulting programs making them quite similar. But these are objective facts which could not be eliminated for educational projects. Unfortunately, from the legal point of view, the reason that some program is marked as plagiarism by some plagiarism detection tool is not enough to reject it. Very often its "author" insists that the program is a product of his or her own and that coincidence of code parts is accidental or is the result of implementing design patterns. The only way to prove that the plagiarism detection tool was right is to demonstrate student that he or she is not aware of "own" code details. In such circumstances there are no sense to find out all common parts.

It should be also noted that programs of the first-year students can be grouped easily by certain errors in the project and/or code structure and other irrelevances. During code borrowing, exactly these unsuccessful elements migrate from one student to other. And this helps instructors to prove that code is plagiarized. Other, and more effective, plagiarism proving method is based upon student's ability to make some minor changes in own code. Its advantage is that it does not suppose an instructor be acquainted with minor details of the disputable submission in advance (instructor can firstly test modified program as "black box"). But this is not a subject of this paper.

This paper discusses the technique which was successfully used in practice and let the author quite effectively figure out code with elements of plagiarism. Or, the dual problem, solved by this technique, is to help an instructor decide if some program is quite original to be positively graded or not and should its author be interviewed or not.

3. Proposed methodology of programming assignment submissions processing

3.1. Experimental data

The main research was conducted in 2018. Its main goal was to formulate a methodology for qualitative estimation of programming assignment code originality or plagiarism. At time of grading, automation of such methodology should help instructors to find out submissions exposing signs of borrowing. At that moment, the author had data consisted of three programming assignments coded in C++ language by approximately 100 students:

1. In a numerical sequence find the maximal subsequence that satisfies some condition.
2. Implement certain traversal of some area of a square matrix.
3. Build class which models some device, then build this class client code so that a user can manage this device interactively.

First two tasks were more concerned about algorithms, but the third one dealt mainly with code organization. Individual tasks for each assignment were similar, but not identical. They could use some universal decisions for certain parts (for example, sequence input), and these common parts of code were not separated at time of submissions processing (and this approach is opposed to [6]). Also, our teaching staff never restricts students to using fixed IDE for code developing.

In spring 2020, the programming course was concerned about Python. Due to specifics of distance learning, we should refine decision rules. As experimental data were used programming assignment about processing text file with some table data. Thus, the input consists of all the submissions of some programming assignment. (Different programming assignments are processed separately.) Let us denote the input as D and suppose there are K submissions. Without loss of generality, suppose $D = \{s_1, s_2, \dots, s_K\}$, where each submission s_i is a set of text files with program code.

3.2. Short description of the algorithm

The final algorithm has three main logical steps. At step 1 each submission is transformed into a set of n-grams. To implement this, some text preprocessor, as well as n-gram length (i.e., value of n), should be specified. Let $grams(s, n)$ denote the set of n-grams constructed for submission s.

Step 2 deals with all the submissions converted to sets of n-grams. Let us define the intersection number of submissions s' and s'' as the number of elements in intersection of their corresponding sets of n-grams and denote it as $s' \otimes s''$. By definition,

$$\mathrm{s}' \otimes \mathrm{s}'' = |\mathrm{grams}(\mathrm{s}', \mathrm{n}) \cap grams(\mathrm{s}'', \mathrm{n})| \quad (1)$$

At step 2 intersection numbers are calculated for each pair of submissions. For each submission, in the output of step 2 there is a multiset of its intersection numbers with the other submissions.

At step 3 constructed multisets are tested. The key points of these processes will be described in detail later.

3.3. Step 1: transforming submissions into sets of n-grams

Transforming submissions into sets of n-grams involves given text files preprocessing. There are different approaches to the problem what should be considered as a symbol while n-gram forming: a word (i.e., token) or literally a symbol of the text. A chosen approach and its details can be encapsulated in a subroutine which implements code preprocessing.

Subroutines *A* and *B* were selected as preprocessing transformations for later analysis.

Subroutine *A* eliminates comments, white symbols, and string literals (except f-strings for Python) from code. Disposing of literals and comments (as containing natural language elements) let us slightly reduce amount of main memory needed for storing all possible *n*-grams. But the identifiers are essential to our goal. Students usually do not change them in borrowed code, and, nevertheless, if they do, then these changes are hardly noticeable. Another reason is that those who use borrowed code without understanding do not feel themselves free to update identifiers properly, i.e., saving meaningful names.

As for white symbols, atypical same spacing could help see borrowings more definitely. Nevertheless, analyzing spaces in code is only waste of time and memory because many modern IDEs format code automatically. Subroutine *B* converts text into a programming language token sequence. This transformation converts all number literals into the same token as well as all identifiers. Having token numbered, the resulting sequence is a sequence of corresponding token numbers. From technical point of view, our implementation casts these numbers to characters.

Let X be preprocessing transformation, then for any file f let us denote the result of applying subroutine X to f as $X(f)$. The value of $X(f)$ is a string anyway.

Let $grams(w, n)$ denote the set of all n-grams of string $w = a_1 a_2 \dots a_m$. The definition from [1] can be formally written in our notation in a such way

$$\mathrm{grams}(w, n) = \{a_k a_{k+1} \dots a_{k+n-1} \mid k \in \overline{1, m-n+1}\} \quad (2)$$

For clearness, it should be stated that a set (not a multiset) is considered. Then, assuming preprocessing subroutine *X* is chosen, let us form the set of *n*-grams of submission from all *n*-grams of its files. Formally, let submission s contain files $f_1, f_2, \dots, f_t$, i.e., $s = \{f_1, f_2, \dots, f_t\}$, then

$$grams(s, n) = \bigcup_{i=1}^{t} grams(X(f_i), n) \quad (3)$$

3.4. Adjustment of n-gram length

During preliminary analysis, different n-gram length values were examined. Our main goal was to find balance between quality of plagiarism determination, optimization of the space complexity of analyzer and time for analyzer code development. It was noticed that at transition from $n = 5$ to $n = 6$ the number of n-grams increases not so drastically as before. Therefore, it was suggested that next small increase of n-gram length would not change the general estimations of originality/similarity qualitatively. Clear, that given a large value of n, many partial borrowings could be skipped. Also, a small value of n leads to large intersection numbers regardless of similarity presence.

Finally, the decision was to use value $n = 5$ for subroutine *A*. For subroutine *B*, the choice was $n = 9$ on the same considerations about increasing the number of n-grams. Let us describe one more reason. The if, for, while, and other statements like that can be found almost in every program and these

statements have specific syntax and a lot of standard usage cases. That is why small values of n can result in plenty of false positive results in case of token sequences analysis by n-grams methods. The trigram method, described in [2], is not the best one for program code. It is obvious that two different almost trivial functions can produce the same sequence of programming language tokens. For similar programs, most corresponding short if statements produce very similar token sequences as well. One can state the same about other types of statements.

3.5. Step 2: calculation of intersection numbers

Now, for each submission from D (the set of the submissions), we should compute its intersection numbers. More formally, multiset $\{s \otimes s' | s' \in D \setminus \{s\}\}$ should be obtained. For step 3 (analysis), it is convenient to store such multiset as a nonincreasing sequence of its elements. Let $a(s)$ denote such sequence for submission s. The output of step 2 consists of such sequences for the given submissions.

3.6. Analysis of obtained data

The main idea of determining if a given submission is original is based upon density of its largest intersection numbers. Before formulating the decision rule, let us see some observations.

In Table 1 and Table 2 for the submission about which is known for certain that it is not original, the greatest intersection numbers, their dynamics and corresponding Jaccard similarity values are presented. It should be noted that the greatest intersection numbers for 5-grams and 9-grams were achieved on the different pairs of packages.

Table 1

The largest intersection numbers in case of borrowings presence for $n = 5$ and preprocessing subroutine A

Intersection number	Distance to the nearest less intersection number	Jaccard similarity
680	206	0.42
474	14	0.24
460	6	0.24
454	2	0.21
452	3	0.20
449	7	0.21
442	13	0.22
429	7	0.21
422	1	0.20
421	9	0.21
412	1	0.21

As it turned out, for packages with borrowings, the distance between the greatest value and next to it was substantially more than other distances between successive intersection numbers. Also, there were situations when substantial decreasing of distances, i.e., some jump, took place somewhere on the fifth largest value. Such situation is possible if an almost identical code is submitted not by two but by the greater number of students. Tables 3 and 4 present the same data for the submission about which is known for certain that it is original. One can see that for original submission nearby distances between the largest values after preprocessing with subroutine A differ far less than for submissions with plagiarism. After preprocessing with subroutine B these distances can be almost stable.

3.7. Step 3: making a decision

In spring 2018, the decision about originality of submission s was made after analyzing the first components of $a(s)$ (with $X = A$ and $n = 5$).

Suppose $a(s) = (a_1, a_2, \ldots, a_{K-1})$ and s_i is a submission which corresponds to intersection number a_i (i $\in \overline{1, K-1}$). (Recall that sequence $a_1, a_2, \ldots, a_{K-1}$ is nonincreasing.)

Table 2
The largest intersection numbers in case of borrowings presence for $n = 9$ and preprocessing subroutine B

Intersection number	Distance to the nearest less intersection number	Jaccard similarity
197	12	0.29
185	16	0.30
169	24	0.22
145	2	0.21
143	5	0.18
138	7	0.21
131	1	0.18
130	1	0.16
129	2	0.17
127	1	0.14
126	1	0.18

Table 3
The largest intersection numbers in case of original submission for $n = 5$ and preprocessing subroutine A

Intersection number	Distance to the nearest less intersection number	Jaccard similarity
480	19	0.24
461	5	0.23
456	2	0.23
454	20	0.24
434	17	0.21
417	1	0.19
416	4	0.20
412	2	0.18
410	1	0.19
409	5	0.19
404	2	0.20

If distance between two largest intersection numbers was more than 15% of the submission greatest intersection number, i.e., $(a_1 - a_2)/a_1 > 0.15$, then this submission was marked as suspicious. Submission s_1, on which the greatest intersection value was achieved, was marked as suspicious too. In some cases, submission s_1 did not fulfill criterion of 15%, but the value of $(a_1 - a_2)/a_1$ was quite significant anyway. If this distance was less than 15% of the greatest intersection number, then, in addition, dynamics of the ten largest intersection numbers $a_1, a_2, \ldots, a_{10}$ was considered in search of a jump. At average, intersection numbers for $X = B$ and $n = 9$ was less than the same for $X = A$ and $n = 5$. Thus, the jumps of the nearby distances were not so obvious in case of tokens processing (subroutine *B*) as in case of single characters processing (subroutine *A*). Investigation of numerical series was limited to the first ten largest elements on considerations that code borrowings were not total among students and that among approximately 100 students from different academical groups of different lab instructors, hardly more than 10 students simultaneously would take the same source. Examination of the greatest intersection numbers obtained for $X = A$ and $n = 5$ allowed effectively find out submissions with borrowings: suspicious submissions were analyzed by humans and their authors were interviewed. There were almost no false positive results (almost all suspicious submissions were proved to have some plagiarism). As for false negative results, it is impossible to estimate that number exactly due to this process should involve human comparison of all pairs of submission and interview with students. Also, it should be stated, that the number of proved plagiarism cases was approximately 25% greater that it seems to be after code inspection by only humans. From the other side, an instructor usually remembers typical solutions found out in the students' programs. During grading internal program characteristics, no more submissions with plagiarism were found (all the submissions were

reviewed by the same instructor). So, one can suppose that the number of false positive (without plagiarism) results were not too much. The only goal was to determine would be the submission original or not, so estimations were done not quantitively but qualitatively.

Table 4
The largest intersection numbers in case of original submission for $n = 9$ and preprocessing subroutine B

Intersection number	Distance to the nearest less intersection number	Jaccard similarity
149	1	0.21
148	5	0.20
143	1	0.20
142	4	0.19
138	4	0.17
134	1	0.18
133	2	0.20
131	2	0.18
129	1	0.14
128	1	0.17
127	9	0.17

4. Decision procedure refinement

Due to distance learning, in spring 2020 students actively communicate by the internet. As a result, some students coded their individual tasks in small commands. The consequence of this was that the distance from the greatest intersection number to the nearest one more often appears not the largest distance between two neighbor intersection numbers. The main decision-making algorithm should be modified. For submission s and its sequence $a(s) = (a_1, a_2, \ldots, a_{K-1})$, another characteristic was used instead the distance between only two largest intersection numbers. The 11 largest intersection numbers were considered, and a jump was searched. Let

$$\mathrm{j} = \arg\max\{\mathrm{a}_i - a_{i+1} |\ i \in \overline{1,10}\} \tag{4}$$

Then the value of relative accumulated jump, i.e., $(a_1 - a_{j+1})/a_1$, was examined as well as relative jump, i.e., $(a_j - a_{j+1})/a_1$, and relative distance between the two largest intersection numbers, i.e. $(a_1 - a_2)/a_1$. If

$$(a_1 - a_{j+1})/\, a_1 > 0.15, \tag{5}$$

$$(a_j - a_{j+1})/\, a_1 > 0.10, \tag{6}$$

and

$$(a_1 - a_2)/a_1 > 0.10, \tag{7}$$

then submission s was considered as suspicious, corresponding to values $a_1, a_2, \ldots, a_j$ submissions were marked as suspicious too.

If relative accumulated jump is large, but relative jump is not, then here potentially can be situation when some parts was discussed by some academic group in class. So, the condition (6) is designed to except such submissions from the set of suspicious ones. The purpose of the condition (7) is the same. In most cases of borrowing presence, the relative distances between the two first intersection numbers are sufficiently large even if the same code parts are submitted not by two but by more students. (Here we should mention that the last condition should be skipped if student tasks are identical.)

The nearer to the maximal value there is a jump (if any) in the sorted sequence of intersection numbers, the less set of students simultaneously writes such submitted code. The greater is the difference between the nearby greatest values, the more code of the project is borrowed.

Let us note that consideration the first 25% of largest intersection numbers in place of only the first 11 values did not change anything. In the most degree, the same threshold 15% as in 2018 is the result of the same teaching approach and, surely, can have other value for some other assignment.

5. Discussion

It is worth to say some words about Jaccard similarity of submissions. In 2018, Jaccard similarity was calculated for each assignment. Dependency between presence of borrowings and the Jaccard similarity appeared not high. If we analyzed the Jaccard similarity only, then quite much plagiarism would not be found. Processing assignments about maximal subsequence and matrix traversal was uninformative. No valid programs with signs of borrowings were found. Using borrowed main code for the algorithmic assignment is almost impossible: needed on its modification time and skills are much more significant than development from the very beginning. The main attention was devoted to the assignment about device modelling. This assignment is more complicated in the sense that it needs some code organization and incremental design to make the program works right.

For device modelling programs, pairwise calculation of Jaccard similarity showed very surprisingly results. It appeared that means of Jaccard similarities for 5-grams after subroutine *A* applying and for 9-grams after subroutine *B* applying were approximately 0.15 and 0.105. At the same moment, the greatest values of similarity were 0.67 and 0.91 accordingly. But there were unoriginal submissions with maximal similarity to the other packages about 0.21 as well as original ones with maximal similarity to the other packages about 0.24 (after preprocessing subroutine *A*). The same situation was in case subroutine *B* applying. It was impossible to settle any Jaccard similarity threshold for all pairs to separate original submissions from plagiarized ones. This phenomenon is a consequence of assignments similarity, discussions that took place in classes, and varying packages length.

Also, some experiments for subroutine A were conducted in 2020. It was interesting what influence increasing the value of n from 5 to 6 would have. 21 suspicious packages were found with the threshold 15% for a relative accumulated jump and $n = 5$. After changing to $n = 6$, the number of suspicious packages became 28. From 21 suspicious packages only one was not marked as suspicious after 6-grams analysis. The packages marked suspicious only with $n = 6$ were examined not only by programs. It turned out that threshold 15% for $n = 6$ was not accurate, and the result was approximately equal to the computation with threshold 11% and $n = 5$. Having threshold 11% for relative accumulated jump, we should compare the values of relative jump and relative distance between the two first intersection numbers not with 0.10 but with less values. Three discussed thresholds can depend not only on the nature of the problems and their parts discussed in classes but on *n*-gram length. The largest intersection numbers are slightly reducing with growth of n so the thresholds should be changed.

Like neural networks are learned on data, after adjusting the value of n, the thresholds can be adapted for each programming assignment personally. Assuming that the vast majority of students do not use borrowed code, one can found out thresholds which cut off only 70%–80% of programs as original. (The percentage should be a little less than expected percent of original programs.) Then their values can be gradually reduced with simultaneous code inspection of some "new" suspicious submissions and their matches. If such inspection discovers no code that looks as borrowed, then thresholds are found (the latest reducing of the thresholds should be discarded). Remember, that the final decision is made after interview with a student. So, too small thresholds can lead to interviewing almost all students that could be unacceptable due to office hours limitations. For more accuracy it is possible to build a set of decision rules based upon different preprocessing transformations and discussed relative values as well as apply their AND and OR combinations. The discussed in the paper decision rules work well if cheaters did not use more than one source to borrow significant amount of code. This is true in practice because to compile two or more programs with different structures in one is not easier than to write code without assistance.

6. Conclusions

The described methodology assumes simultaneous processing of all the submissions. The proposed approach is not sensible to mutual location of code parts, to distribution of code between translation units, and to presence of dummy code. It can be easily adapted to different programming languages and their dialects by using preprocessing routines appropriate to a language. Moreover, it is acceptable that some parts of assignment are discussed in class or contain pieces of code provided by instructors. An important point here is that these parts should not be identified and removed before processing.

Also, the described technique is quite agile in the sense that can be specialized not only with preprocessing subroutine and *n*-gram length, but with thresholds for decision-making rules. This methodology can be used with slightly different tasks of programming assignments as well as with identical ones. The better results are exposed on quite complex tasks which need thoroughly code organization and incremental design.

The proposed technique only helps instructors to find out plagiarism among students but gives no guarantees. The same thing is true for other methods in this field. In any case there could be some students who successfully submit third-party code or code with elements of plagiarism. To reject original code is impossible due to plagiarism proving procedure moderated by instructors.

Further research can deal with applying the proposed density analysis not to intersection numbers but to other pairwise similarity measures. The analogous situation with density of pairwise similarity series is expected. But there are strong doubts that a lot of time wasted on programming more sophisticated pairwise characteristics gives enhancement adequate to coding time or gives any enhancement at all. Highly likely, it will be pure theoretical research. Another branch of research can deal with code compiled from two or more sufficiently dissimilar sources.

7. References

[1] J. Leskovec, A. Rajaraman, J. Ullman, Mining of massive datasets, 2nd ed., University Press, Cambridge, 2014.

[2] MAC Jiffriya, MAC Jahan, R.G. Ragel, S. Deegalla, AntiPlag: plagiarism detection on electronic submissions of text based assignments, in: Proceedings of the 8th International Conference on Industrial and Information Systems, ICIIS 2013, IEEE, NJ, 2013, pp. 376-380. doi:10.1109/ICIIS31010.2013.

[3] B. Minaei-Bidgoli, M. Niknam, An n-gram based method for nearly copy detection in plagiarism systems, in: Proceedings of the 6th Workshop on Awareness and Reflection in Technology Enhanced Learning (ARTEL 2016), Lyon, France, September 13, 2016.

[4] A. Barrón-Cedeño, P. Rosso, On automatic plagiarism detection based on n-grams comparison, in: M. Boughanem, C. Berrut, J. Mothe, C. Soule-Dupuy (Eds.), Advances in Information Retrieval, ECIR 2009, volume 5478 of Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg, 2009, pp. 696-700.

[5] D. Grune, M. Huntjens, Het detecteren van kopieën bij informatica-practica. Informatie 31(11) (1989): 864-867.

[6] L. Prechelt, G. Malpohl, M. Philippsen. Finding plagiarisms among a set of programs with JPlag. J Univers. Comput Sci. 8(11) (2002): 1016-1038.

[7] D. Fu, Y. Xu, H. Yu, B. Yang. WASTK: An weighted abstract syntax tree kernel method for source code plagiarism detection. Scientific Programming (2017). doi:10.1155/2017/7809047.

[8] H. Cheers, Y. Lin, S.P. Smith. Academic source code plagiarism detection by measuring program behavioral similarity. IEEE 9 (2021): 50391-50412. doi: 10.48550/arXiv.2102.03995.

[9] V. S. Anjali, T. R. Swapna, B. Jayaraman. Plagiarism detection for Java programs without source codes. Procedia Computer Science 46 (2015): 749-758.

[10] J.-H. Ji, G. Woo, H.-G. Cho. A plagiarism detection technique for Java program using bytecode analysis, in: 2008 Third International Conference on Convergence and Hybrid Information Technology, 2008, pp. 1092-1098. doi: 10.1109/ICCIT.2008.267.

[11] O. Karnalim. Python source code plagiarism attacks on introductory programming course assignments. Themes in Science and Technology Education, 10(1), 2017, pp. 17-29.

[12] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, B. Maqbool, A Systematic Review on Code Clone Detection, in: IEEE Access, 2019, vol. 7, pp. 86121-86144. doi: 10.1109/ACCESS.2019.2918202.

[13] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, S. Chakraborty. Towards learning (dis)-similarity of source code from program contrasts, in: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, May 22-27, 2022, v. 1, pp. 6300-6312.

[14] J. Yasaswi, S. Purini, C. V. Jawahar. Plagiarism detection in programming assignments using deep features, in: 2017 4th IAPR Asian Conference on Pattern Recognition (ACPR), 2017, pp. 652-657. doi: 10.1109/ACPR.2017.146.