computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

PROGRAMMING STYLE ON SOURCE CODE PLAGIARISM AND COLLUSION DETECTION

Oscar Karnalim, Gisela Kurniawati

Maranatha Christian University, Prof. drg. Surya Sumantri, M.P.H. Street no 65, Bandung, 40164, Indonesia,
oscar.karnalim@it.maranatha.edu, https://orcid.org/000-0003-4930-6249, gisela.kurniawati@it.maranatha.edu

Paper history:
Received 12 August 2019
Received in revised form 7 December 2019
Accepted 21 February 2020
Available online 31 March 2020

Keywords:
source code;
plagiarism and collusion;
similarity detection;
programming style;
computing education.

Abstract: This paper utilises programming style on a source code plagiarism and collusion detection to both capture obvious attempts of such academic dishonesty (which characteristics are ignored on most detection techniques) and prioritise non-coincidental similarity to the coincidental one (as only the former can raise suspicion). The technique relies on pairwise programming style similarity to deal with the former and dishonesty probability (how significant is the programming style change between the author's current submission and previous submissions) to deal with the latter. According to our evaluation, programming style similarity can increase precision since when a code is copied, the programming style can be unconsciously shared (especially for novice students). Dishonesty probability increases not only precision but also recall, f-score, and the resulted similarity degree of suspected pairs; the copied code commonly has different programming style in comparison with the student's usual style (captured from previous submissions). Our detection technique is comparable to a common technique in academia except that it takes longer processing time as more hints are generated and considered.

1. INTRODUCTION

Source code plagiarism and collusion occur when source code is copied and reused with inadequate acknowledgment toward the original authors [1], [2]. They are only different in a sense that on the former, the original authors are not aware about it. Both are emerging issues in computing education. They do not only lead to unfair grading [3] (in which plagiarists and colluders get the score they do not deserve) but the institution reputation is also at stake [4] (as student grades do not reflect the programming skills).

Several preventive approaches have been developed to deal with source code plagiarism and collusion [5], where some of them consider student motivation in doing such illegal behaviours [6]. Educating the students about what are encouraged [7] and utilising a similarity detection tool (such as JPlag [8]) to raise suspicion for plagiarism and collusion are possibly two of the most practical ones.

A similarity detection tool commonly pairs student works and alerts the lecturers if some (or all) pairs share high similarity [9]. The pairs are then observed further by the lecturers to assure whether the cause is plagiarism or collusion (as high similarity does not always entail academic misconduct [10]).

Most techniques applied in the similarity detection tools remove non-semantic-preserving information (e.g., comments and whitespaces) prior comparison. We are aware that this prevents trivial disguises in altering source code similarity. However, the similarity of this information can be a strong hint for source code plagiarism and collusion [11]. A pair of source code files with the same program structure, for instance, will become more suspicious if they also share the same comments.

Another limitation of those techniques is that no distinction between coincidental and non-coincidental similarity, even though only the latter can raise suspicion for plagiarism or collusion. This can be labour-intensive for lecturers as that distinction should be done manually.

In response to the aforementioned issues, this paper proposes a detection technique that considers style similarity and dishonesty probability in addition to content similarity. Style similarity captures the similarity of non-semantic-preserving information (represented as programming style). Dishonesty probability prioritises non-coincidental over coincidental similarity by giving higher degree to the former, assuming it occurs when the students suddenly change their programming style. The impact of those two are set far smaller than content similarity as we believe the content similarity should still be prioritised in capturing the culprits.

This paper aims at reducing the number of false results, suspected code pairs that are not resulted from plagiarism and/or collusion, by differentiating the coincidental from the non-coincidental ones via style and dishonesty probabilities, derived from the programming style. To the best of our knowledge, the technique is the first of this type.

2. RELATED WORKS

In academia, most source code similarity detection techniques convert given source code into intermediate representation [12], a concise format that contains only important information for comparison. The representation varies from source code attributes to program dependency graph.

Early techniques relied on source code attributes in which most of them are indirectly related to program semantic. A technique proposed in [13] is arguably the oldest one for this category. It determines similarity based on the number of operators, operands, unique operators, and unique operands. Many follow-up techniques were then developed upon that, which details can be seen in [14].

Instead of superficial attributes, several techniques introduced source code token string, an array-like representation storing source code tokens based on their occurrence order [15]. This representation is resistant to comments and whitespace modification as those two are removed prior comparison. The similarity is often calculated with a string matching algorithm, such as Running-Karp-Rabin Greedy-String-Tiling (RKRGST) [8], [12], [15], [16] or string alignment [17], [18].

The similarity degree of source code token string is often affected by syntactical change (e.g., replacing a for loop with a while loop). Therefore, low-level token string was introduced [19]. This is similar to source code token string except that it is resulted from the compiled form of the source code. As such a form is often optimised, many program statements are converted into their semantic, which automatically handles the syntactical change. In terms of similarity measurement, it uses string matching algorithms: RKRGST [20], [21] or string alignment [18].

Relying on a string matching algorithm can cause time deficiency issues as most string matching algorithms (RKRGST and string alignment in particular) are not linear time. Consequently, several techniques relied on Information Retrieval (IR) measurement [22], which is generally fast to compute since each token string is considered as a bag of words. These techniques used either Latent Semantic Analysis [9], [23], [24] or BM25 [25].

Some techniques [12], [26], [27] utilised an IR measurement to filter source code pairs inputted to string matching algorithm; only pairs which share high IR similarity are further compared with the string matching algorithm. This should be more token-order-sensitive than standard IR techniques but still more time efficient than standard string matching techniques.

Abstract syntax tree, parse tree, and program dependency graph can also be used as alternatives to gain more semantic information. However, since comparing those representations can be time consuming due to their abstract nature, several heuristics are often applied for either comparing the code files (e.g., kernel methods in [28], [29]) or filtering comparison candidates (such as the one proposed in [30]).

Instead of applying heuristics in comparing abstract syntax tree, a technique in [31] converts the syntax trees to token strings through pre-order visit and therefore compares them with a string matching algorithm. This was followed by [32] but with hashing mechanism involved.

Most techniques agree that program semantic is important on determining similarity. Hence, they remove all non-semantic-preserving information. This can be beneficial for dealing with trivial similarity disguises as most of them do not affect the program semantic. Nevertheless, removing it completely means such information cannot be used as a hint for plagiarism and collusion, even though they are strongly convincing if found [11]. For example, two source code files with the same content will become more suspicious if they share the same layout.

Regarding this matter, a technique in [11] incorporated comment and inconsistency similarity in addition to content similarity. Comment similarity is measured by extracting all the comments in trigram format and performing the comparison with Cosine Correlation. Inconsistency and content similarities are measured in the same way except that their features are unusual patterns and trigram source code tokens respectively. All similarity scores are then merged as a final score through their own defined equation.

Comment similarity was also used in [33] in combination with variable type and content similarity. Unique to this, each similarity factor is determined with a specific measurement: comment similarity relies on a string matching algorithm, variable type similarity relies on Cosine Correlation, and content similarity relies on program dependency graph comparison.

A technique in [34] applied a five-staged string matching based comparison, in which some of the stages consider comments and/or whitespaces. It starts processing given source code files as their original forms and ends with their source code token strings. This staging process was also followed by [35] but with a different set of stages.

Whitespace occurrences were also considered in [36] along with the statistics of code layout, delimiter, identifier, and keyword. On that technique, the similarity of such features (measured with weighted mean) was displayed separately from content similarity (measured with a string matching algorithm).

Another issue is that most detection techniques perceive coincidental and non-coincidental similarities as the same. This can lead to a larger amount of manual work for lecturers as only non-coincidental similarity can lead to plagiarism or collusion.

We believe few detection techniques can mitigate such an effort for distinguishing even though it was not one of their primary aims. A technique in [37] for instance, can reduce the number of pairs with coincidental similarity during an in-class offline assessment as it limits the suspicion on source code files which authors are adjacently seated. Another example is a work proposed by [38] which timestamps each save actions and embeds it on the code. If a source code pair shares the same timestamps, it can be assured to contain non-coincidental similarity.

3. METHOD

This paper proposes a similarity detection technique that considers non-semantic-preserving information, which is programming style in our case. Further, it prioritises pairs with non-coincidental similarity via an assumption that such similarity can involve a significant change of programming style (as at least one code file is not created by the same author).

Compared to other detection techniques which consider non-semantic-preserving information [11], [33]–[36], our technique is argued to be more sensitive to such information as its coverage is not limited to comments and whitespaces [11], [33–[35] and it relies on many features (83 features in total while the highest number of features used in existing techniques is only 17 [36]).

Our technique is also more practical to be used compared to other techniques that are able to distinguish coincidental and non-coincidental similarity. It requires no specific assessment constraints [37] and IDE [38].

The technique accepts Java source code files as its input (but can accommodate other languages with some adjustments) and works in threefold. At first, the code files are paired one another. To illustrate this, if the code files are *code1*, *code2*, and *code3*, the pairs will be *code1*-*code2*, *code1*-*code3*, and *code2*-*code3*.

Secondly, the similarity degree for each pair is calculated as in (1) by default; $csim$ is content similarity covering semantic-preserving information, $ssim$ is style similarity covering non-semantic-preserving information, and $dprob$ is dishonesty probability differentiating non-coincidental from coincidental similarity based on programming style change toward previous submissions. The details of how each component is calculated will be described later

$$s = 0.8 * csim + 0.1 * ssim + 0.1 * dprob, \quad (1)$$

where *csim* is given the largest proportion in determining the final similarity since we believe that shared semantic-preserving information should still be the main reason for capturing the culprits.

As *dprob* relies on previous submissions, it is not suitable to be used at the beginning of a course. On that situation, *dprob* can be ignored, resulting (2)

$$s = 0.9 * csim + 0.1 * ssim. \quad (2)$$

Thirdly, all code pairs which similarity degree is higher or equal to the maximum of 75% and the mean of similarity degree will be considered as suspicious. The threshold is determined in such a way to assure that the suspected pairs share high similarity (at least 75%), and if most pairs share high similarity, the pairs are limited to those which similarity is unusually high.

Content similarity (*csim*) refers to how much semantic-preserving information shared between two source code files. It is measured by converting both code files to token strings using ANTLR [39] (with comments and whitespaces removed). After that, 4-gram tokens are extracted from the strings. *n*-gram token is a concatenated form of extracted tokens, which considers *n* adjacent tokens as one. In this case, each token is formed by considering four adjacent tokens as *n*=4. Finally, the resulted 4-gram token sets of both code files are compared with Cosine Correlation in Vector Space Model [22].

Style similarity (*ssim*) is the counterpart of content similarity. It measures how much non-semantic-preserving information (as programming style features) shared between two source code files using Cosine Correlation in Vector Space Model [22]. Each code file is converted to a vector covering four style features (comment content, identifier name, code layout, and program flow keyword, with 83 features in total) prior comparison.

Comment content features describe how the author chooses words to express their intention on comments. The features can be seen in Table 1. Top-5 *n*-gram comment words used in *com01-com20* refer to five *n*-gram comment words which occurrence are the most common on given code pair; each *n*-gram word is formed by concatenating *n* adjacent words. Comment words are generated by tokenising the comments based on whitespaces and punctuation, in which each word is then lowercased and stemmed with Porter Stemmer. If some of the words look like identifiers with concatenated subwords, these words will be split further using a domain-specific tokenisation [40] prior lowercasing and stemming. This, for example, will convert *MusicController* to *music* and *controller*. *com21-com24* are about the proportion of a particular type of character, normalised toward the total number of comment characters including whitespaces. *com25* is about the average length of comment words.

Identifier name features describe how the author selects words to form their identifiers. The features can be seen in Table 2. *idn01-idn20* are similar to *com01-com20* except that they rely on identifier words instead of comment words. Identifier words are generated by splitting each identifier with a domain-specific tokenisation [40] in which the results are then stemmed with Porter Stemmer [22]. *idn21-idn24* are about the proportion of a particular type of character, normalised toward the total number of identifier characters. *idn25* and *idn26* are about the average length of identifiers and identifier words respectively. The former is the input of the splitting mechanism while the latter is the output.

Code layout features describe how the author incorporates whitespaces (space, tab, and newline) while creating the code. The features can be seen on Table 3. Layout tokens for *lay01-lay05* are extracted from a source code token string that accentuates the impact of whitespaces. That string is generated using ANTLR [39] but with whitespaces recognised as tokens and the counterparts anonymised. *lay06-lay11* are about the proportion of a particular type of character, normalised toward the total number of source code characters including whitespaces.

Table 1. Comment Content Features for Style Similarity

ID	Description
com01-com05	The occurrence frequency of top-5 1-gram comment words from given code pair, normalised by the total number of words.
com06-com10	The occurrence frequency of top-5 2-gram comment words from given code pair, normalised by the total number of words.
com11-com15	The occurrence frequency of top-5 3-gram comment words from given code pair, normalised by the total number of words.
com16-com20	The occurrence frequency of top-5 4-gram comment words from given code pair, normalised by the total number of words.
com21	Comment vocal proportion toward the total number of comment characters including whitespaces.
com22	Comment consonant proportion toward the total number of comment characters including whitespaces.
com23	Comment digit proportion toward the total number of comment characters including whitespaces.
com24	Comment punctuation proportion toward the total number of comment characters including whitespaces.
com25	Average length of comment words.

Table 2. Identifier Name Features for Style Similarity

ID	Description
idn01-idn05	The occurrence frequency of top-5 1-gram identifier words from given code pair, normalised by the total number of words.
idn06-idn10	The occurrence frequency of top-5 2-gram identifier words from given code pair, normalised by the total number of words.
idn11-idn15	The occurrence frequency of top-5 3-gram identifier words from given code pair, normalised by the total number of words.
idn16-idn20	The occurrence frequency of top-5 4-gram identifier words from given code pair, normalised by the total number of words.
idn21	Identifier vocal proportion toward the total number of identifier characters.
idn22	Identifier consonant proportion toward the total number of identifier characters.
idn23	Identifier digit proportion toward the total number of identifier characters.
idn24	Identifier punctuation proportion toward the total number of identifier characters.
idn25	Average length of identifiers.
idn26	Average length of identifier words.

Program flow keyword features describe how the author chooses program flow keywords in solving the problem. The features can be seen in Table 4. For *pro01-pro15*, structure tokens are extracted from a source code token string that accentuates the impact of program structure. The string replaces all identifiers with the same token called *ident*, and removes all whitespaces, brackets, and semicolons. *pro16*-*pro21* are about the proportion of a particular program flow keyword toward all keywords used on the code file.

Table 3. Code Layout Features for Style Similarity

ID	Description
lay01-lay05	The occurrence frequency of top-5 4-gram layout tokens from given code pair, normalised by the total number of such tokens.
lay06	Semicolon proportion toward the total number of source code characters including whitespaces.
lay07	Curly bracket proportion toward the total number of source code characters including whitespaces.
lay08	Standard bracket proportion toward the total number of source code characters.
lay09	Space proportion toward the total number of source code characters including whitespaces.
lay10	Tabulation proportion toward the total number of source code characters including whitespaces.
lay11	Newline proportion toward the total number of source code characters including whitespaces.

Table 4. Program Flow Keyword Features for Style Similarity

ID	Description
pro01 - pro05	The occurrence frequency of top-5 2-gram structure tokens from given code pair, normalised by the total number of such tokens.
pro06 - pro10	The occurrence frequency of top-5 3-gram structure tokens from given code pair, normalised by the total number of such tokens.
pro11 - pro15	The occurrence frequency of top-5 4-gram structure tokens from given code pair, normalised by the total number of such tokens.
pro16	*If* keyword proportion toward the number of keywords.
pro17	*Switch* keyword proportion toward the number of keywords.
pro18	*While* keyword proportion toward the number of keywords.
pro19	*Do-While* keyword proportion toward the number of keywords.
pro20	*For* keyword proportion toward the number of keywords.
pro21	*Try* keyword proportion toward the number of keywords.

Dishonesty probability (*dprob*) refers to how confident a source code pair can be considered to contain non-coincidental similarity, assuming that such similarity occurs when the authors' programming style is suddenly changed. This is calculated as in (3) where *dauth(code)* refers to disauthorship probability, depicting how convincing the parameterised *code* has different programming style compared to the author's previous submissions.

Disauthorship probability is determined with the help of Multinomial Naive Bayes (MNB) [41] in which each author is exclusively assigned with one MNB learning model, built from the author's previous submissions. Each submission is converted to style features (which are also used for calculating style similarity). However, *com01-com20*, *idn01-idn20*, *lay01-lay05*, and *pro01-pro15* consider all code files written by that author instead of only a code pair. In our case, MNB is implemented with WEKA [42]

$$dprob = \frac{dauth(code1) + dauth(code2)}{2}. \quad (3)$$

On each learning model, the target class is binary, representing whether one code file's programming style is similar to the author's style. To represent code files which style is different from the author's, an equal number of random source code files taken from other students will be used.

Disauthorship probability is generated in twofold. First of all, style features from the author's currently-submitted code file are generated and then fed to the author's learning model for classification. Such classification returns two probabilities: a probability suggesting the same programming style and a probability suggesting the counterpart. Commonly, the classification result is determined by the highest probability. However, that step is ignored in our case and disauthorship probability is assigned with a probability suggesting different programming style.

It is important to note that style similarity and dishonesty probability can also be treated as separate metrics for further hints in suspecting source code plagiarism and collusion. A pair with suspiciously high style similarity is an obvious attempt of such academic dishonesty. Dishonesty probability, even though it is only applicable after several submissions, can assure that a pair shares non-coincidental similarity once the programming style is suddenly changed.

Disauthorship probability (a part of dishonesty probability) can also be used to distinguish the culprits from the victims if high content similarity is on board. Such distinction can be used to educate them separately [37] or setting the appropriate penalty (if not all dishonesty acts are considered equal). Let us assume we have a code pair with high content similarity in which A and B are the students. If A's code results in high disauthorship probability while B's does not, it can be stated that B is the original creator of the code and A has copied the code. This also works in reverse if B's code is the one which results in high disauthorship probability. If both student code files result in high disauthorship probability, then they copy the code from another student. It can also be a potential case for ghostwriting in which the students ask someone to help (who is commonly an expert in programming) to do their assignments. If both student code files result in low disauthorship probability, the case is convinced to be collusion instead of plagiarism; the students may create the code together.

4. EVALUATION

The proposed technique was evaluated under six research questions:

R1 How effective is style similarity in capturing non-semantic-preserving information?

R2 What is the impact of style similarity on detection performance?

R3 How effective is Multinomial Naive Bayes (MNB) in classifying programming style?

R4 Which style features do affect the accuracy of MNB?

R5 What is the impact of dishonesty probability on detection performance?

R6 How effective and efficient is our proposed technique with both style and dishonesty probabilities on board compared to a common technique in academia?

To address those questions, three datasets were introduced: artificial, introductory, and historical dataset. Artificial dataset was created by an author in which the differences between code files are focused on non-semantic-preserving information relating to programming style. It covers four categories: comment content (how the comment words are chosen), identifier name (how the names are formed), code layout (how the whitespaces are used), and program flow keyword (how some keywords are preferred to others).

We used the same Java original code files as in [20], describing seven introductory programming materials: output (T1), input (T2), branching (T3), looping (T4), function (T5), array (T6), matrix (T7) [43]. On each category, the original files were then copied and modified by a particular treatment. For comment content category, a descriptive comment is added for each instruction. For identifier name category, all identifiers are renamed. For code layout category, empty lines and tabs are removed. Further, standard brackets on arithmetic expressions are added or removed if possible. For program flow keyword category, each program flow statement is replaced with its equivalent statement but with different keyword (e.g., a *for* loop is converted to a *while* loop). Additionally, each main program is encapsulated with a dummy *try-catch* statement. In total, artificial dataset has 28 source code pairs where each category has seven of them.

Introductory dataset contains student works taken from several cohorts of Java introductory programming course. In total, there are 1,028 source code files mapped to 94 assessments (each of them has three to 22 assessments). Per assessment, the suspected pairs were determined using JPlag [8] with average normalisation. To be suspected, the pairs should have similarity degree higher or equal to the maximum value between 75% and the mean of all similarity degree. This threshold is defined with the same reasons as such a threshold on proposed technique. Further, it is also supported by the fact that on each assessment, a pair with the lowest similarity degree was suspicious from an author's perspective and they believe it can be generalised to other pairs with the same or higher similarity degree.

Historical dataset contains 219 Java source code files created by five undergraduate students for solving problems in Open Kattis (*https://open.kattis.com/)*. The students are denoted as *SA*, *SB*, *SC*, *SD*, and *SE* where their number of source code files are 40, 75, 40, 30, and 32 respectively. The suspected pairs were determined in the same way as introductory dataset but all code files were considered as an assessment and each pair in which both code files were created by the same student is excluded.

Three detection techniques were involved in this evaluation. S-VSM and SH-VSM are our proposed techniques; the former relies on content and style similarity while the latter is expanded from the former by considering dishonesty probability. Their similarity calculations can be seen in (2) and (1) respectively. STD-GST is a common technique in academia that converts both given source code files to token strings via ANTLR [39] (with comments and whitespaces excluded), and compares the strings using Running-Karp-Rabin Greedy-String-Tiling (RKRGST) with two as its minimum matching length. This technique is considered as our baseline for evaluating our proposed techniques (S-VSM and SH-VSM).

4.1 THE EFFECTIVENESS OF STYLE SIMILARITY IN CAPTURING NON-SEMANTIC-PRESERVING INFORMATION

Our style similarity is considered as effective in capturing non-semantic-preserving information if for each pair on artificial dataset (which modification is mainly not semantic preserving), S-VSM (our technique which relies on content and style similarity) provides slightly lower similarity degree than a common technique in academia (STD-GST, our baseline that only considers content similarity). This is based on our expectation that such modification should be considered in comparison (to capture obvious attempts of plagiarism or collusion) but with limited impact (so that the culprits cannot obfuscate their academic dishonesty via that modification).

Figure 1 shows that S-VSM generates slightly lower similarity degree than STD-GST toward comment content change. On some cases (T2, T3, T4, and T6), the difference is extremely small, lower than 1%. This is expected since only 10% of the resulted similarity degree comes from style similarity and not all programming style features are related to comment content.

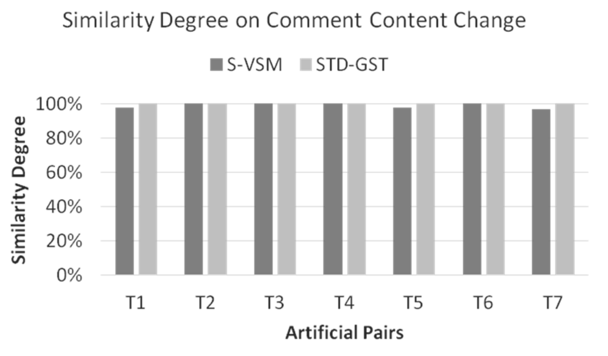

Figure 1 – Resulted similarity degree on pairs with comment content change

In dealing with identifier name change (see Figure 2), both S-VSM and STD-GST are able to capture the change since while measuring content similarity, renamed identifiers are commonly considered as different tokens from their originals. Nevertheless, S-VSM still leads to slightly lower similarity degree than STD-GST since the change also affects style similarity. Further, on S-VSM's content similarity, the impact of such change is propagated through n-gram mechanism; each renamed identifier leads to four mismatched 4-gram tokens [26].

Code layout change, if only affects whitespaces (see T1, T6, and T7 in Figure 3), slightly favours S-VSM even though its difference is lower than 1%; whitespace change is exclusively considered by that technique.

When the change affects other tokens such as brackets (see T2-T5 in Figure 3), both techniques are able to capture the change as these tokens are also considered by content similarity. On half cases (T4 and T5), S-VSM generates slightly lower similarity degree than STD-GST since each additional bracket leads to four mismatched 4-gram tokens at content level and the use of brackets is also considered by style similarity. However on the other half (T2 and T3), this works in reverse. These brackets successfully split some STD-GST's matched token substrings unequally, resulting in the shorter ones not recognisable as matched tokens (their length is shorter than RKRGST minimum matching length).

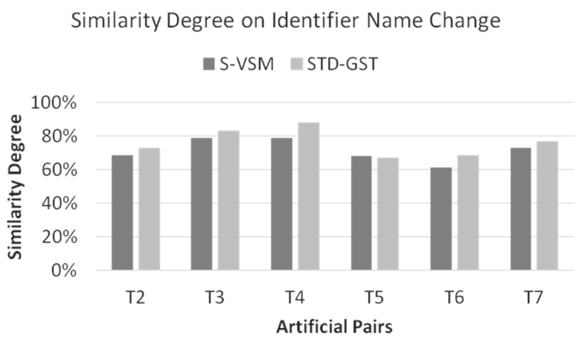

Figure 2 – Resulted similarity degree on pairs with identifier name change

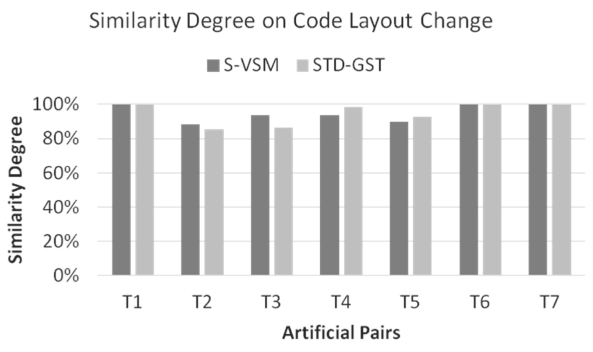

Figure 3 – Resulted similarity degree on pairs with code layout change

Figure 4 depicts that both S-VSM and STD-GST are able to capture program flow keyword change since such change results in altering the program structure and therefore affects content similarity. S-VSM provides slightly lower similarity degree on T4, T5, and T7 due to n-gram mechanism on content similarity and the consideration of keyword usage on style similarity. For remaining cases, it generates higher similarity degree as the change successfully split some STD-GST's matched token substrings unequally, making the shorter ones not recognisable by its matching algorithm.

To sum up, S-VSM is more capable than STD-GST in capturing non-semantic-preserving information; it provides slightly lower similarity degree on most code pairs which difference is mainly focused on that kind of information.

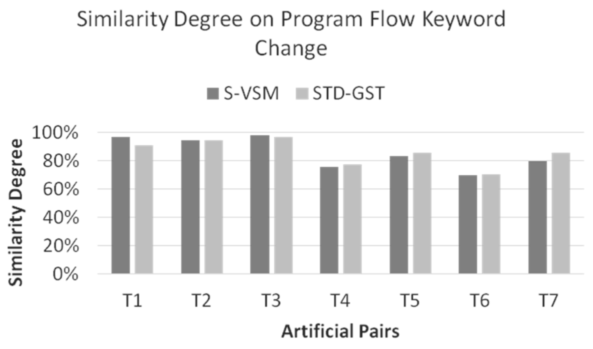

Figure 4 – Resulted similarity degree on pairs with program flow keyword change

4.2 THE IMPACT OF STYLE SIMILARITY ON DETECTION PERFORMANCE

This subsection evaluates the impact of considering style similarity when dealing with semi-real cases of source code plagiarism and collusion detection (i.e., introductory dataset). S-VSM (our proposed technique that considers style similarity) was compared to STD-GST (the baseline technique) with five metrics on board:

- The similarity degree of dataset-suspected pairs checks how effective a technique is in assigning high similarity to suspicious pairs.
- Precision [22] measures how accurate a technique is. It is calculated as in (4) where *dpairs* are dataset-suspected pairs and *tpairs* are technique-suspected pairs

$$P = \frac{|dpairs \cap tpairs|}{|tpairs|}. \quad (4)$$

- Recall [22] measures how sensitive a technique is. It is calculated as in (5) where *dpairs* are dataset-suspected pairs and *tpairs* are technique-suspected pairs

$$R = \frac{|dpairs \cap tpairs|}{|dpairs|}. \quad (5)$$

- F-score [22] is the harmonic mean between precision and recall, calculated as in (6) where *P* is precision and *R* is recall

$$fscore = \frac{2*P*R}{|P+R|}. \quad (6)$$

- Execution time (in nanoseconds) measures how long a technique takes time to process the whole dataset.

S-VSM generates 2.6% lower averaged similarity degree than STD-GST on dataset-suspected pairs (see Figure 5); S-VSM is more prone to modification as it considers style similarity in addition to the content one. Further by nature, S-VSM's content similarity leads to more mismatched tokens due to its n-gram mechanism. Each mismatched content token is a member of four 4-gram tokens and all of these 4-gram tokens are considered as mismatches.

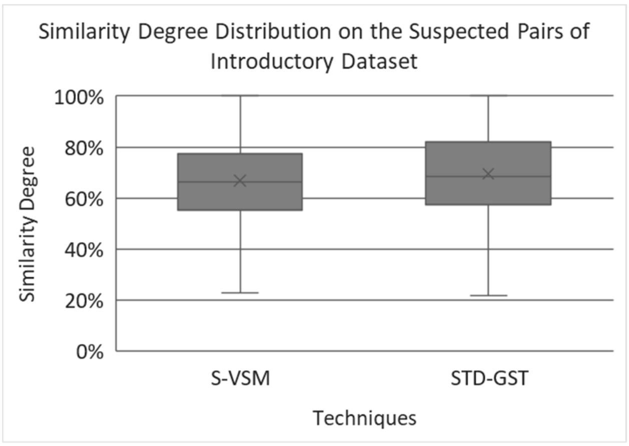

Figure 5 – Resulted similarity degree on the suspected pairs of introductory dataset

As seen in Figure 6, S-VSM results in higher precision than STD-GST (+2.1%) since on the dataset (which students are novice programmers), not all style features on the copied code are changed, and they can provide more hints in determining similarity. That improvement, however, should be exchanged with lower recall and f-score as more factors are introduced on the similarity measurement.

In terms of efficiency, S-VSM can save 30.5% execution time. S-VSM only takes 10.7 seconds to process the whole dataset while STD-GST takes 15.5 seconds.

4.3 THE EFFECTIVENESS OF MULTINOMIAL NAIVE BAYES IN CLASSIFYING PROGRAMMING STYLE

Dishonesty probability used in our detection technique relies on Multinomial Naive Bayes (MNB) in capturing student programming style. Hence, this subsection evaluated the effectiveness of such a classifier with accuracy as the evaluation metric and 10-fold cross validation as the instrument.

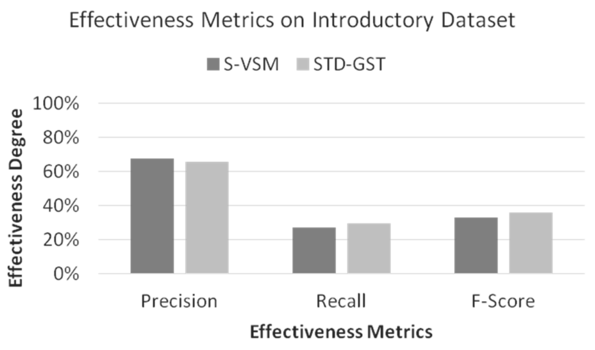

Figure 6 – Effectiveness metrics on introductory dataset

The accuracy of MNB was measured on historical dataset. Per student, their own code files refers to positive instances while an equal number of other students' code files (taken randomly) representing the counterpart. Figure 7 depicts that MNB is considerably effective in capturing programming style with 67.1% as the averaged resulted accuracy. The highest accuracy occurs on *SE* (82.9%) while the lowest one occurs on *SB* (56.7%).

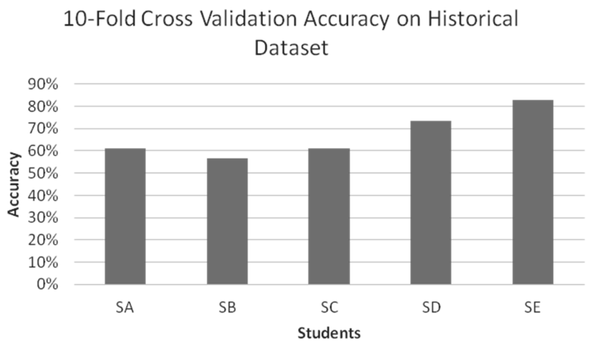

Figure 7 – 10-fold cross validation accuracy on historical dataset

4.4 STYLE FEATURES AFFECTING THE ACCURACY OF MULTINOMIAL NAIVE BAYES

Style features affecting Multinomial Naive Bayes (MNB)'s accuracy were determined with WEKA's information gain [42] under the same setting as previous subsection. Table 5 shows that the number of affecting features varies per student as each student has his own programming style. The lowest one occurs on *SB* (12 features) while the highest one occurs on *SE* (30 features).

Among the affecting features, most top-5 features are related to comments. This is natural since comments can be freely written in natural language without being constrained by programming syntax rules. Other style categories also affect MNB's accuracy. However, their occurrences are not as frequent as the comment ones.

Table 5. Style Features Affecting MNB per Student

Student	Total Affecting Features	Top-5 Affecting Features
SA	14	one of com01-com05, two of com06-com10, com22, and com24
SB	12	one of com01-com05, com22, idn26, one of lay01-lay05, and lay11
SC	22	one of com01-com05, one of com06-com10, com22, com24, and com25
SD	27	two of com01-com05, com21, com24, com25
SE	30	com22, com24, com25, and two of pro11-pro15

In summary, style features affecting MNB's accuracy can vary depending on the student's programming style. However, most of the features are related to comments as free text can be embedded on such part.

4.5 THE IMPACT OF DISHONESTY PROBABILITY ON DETECTION PERFORMANCE

The impact of dishonesty probability was evaluated with two techniques on board: S-VSM and SH-VSM. They are our proposed detection techniques in which the latter is exclusively featured with dishonesty probability. The performance of both techniques were measured on historical dataset – the only dataset in which the author of each code file is known – using five metrics used for measuring the impact of style similarity (see subsection 1.2): the similarity degree of dataset-suspected pairs, precision, recall, f-score, and execution time.

In average, SH-VSM yields slightly higher similarity degree on dataset-suspected pairs compared to S-VSM (see Figure 8 with 1.6% improvement). It also provides higher precision and recall (see Table 6.). Precision is increased by 25% while recall is increased by 0.3%. This therefore slightly increases the f-score by 0.6%. In other words, the use of dishonesty probability (which is exclusive to SH-VSM) makes suspected code pairs become more identical and therefore leads to higher effectiveness, precision in particular.

Integrating dishonesty probability has a drawback of efficiency: SH-VSM takes 3.7 minutes to process the historical dataset while S-VSM only takes about

13 seconds. In addition to creating the learning model, SH-VSM needs time to create the learning vectors per comparison and classify them based on the learning model.

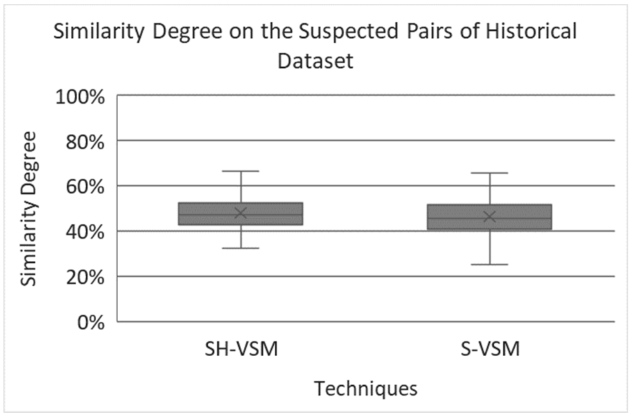

Figure 8 – Resulted similarity degree on the suspected pairs of historical dataset

Table 6 - Effectiveness Metrics on Historical Dataset

Metrics	**SH-VSM**	**S-VSM**	**STD-GST**
Precision	100%	75.00%	100%
Recall	1.2%	0.9%	2.4%
F-Score	2.3%	1.8%	4.7%

4.6 THE PERFORMANCE OF OUR PROPOSED TECHNIQUE COMPARED TO A COMMON TECHNIQUE IN ACADEMIA

This subsection compares our proposed technique that incorporates both style and disauthorship probabilities (SH-VSM) with a common technique in academia (STD-GST) under the same setting as the previous subsection. Table 6 shows that SH-VSM is as effective as STD-GST in terms of precision; both of them lead to 100% precision. However, SH-VSM provides lower recall (-1.1%), f-score (-2.3%), and averaged similarity degree of suspected pairs (-5.2%) as more factors are considered in the comparison.

In terms of execution time, SH-VSM is still slower than STD-GST. The former takes 3.7 minutes while the latter only takes half a minute, processing 15.045 code pairs from historical dataset on Intel Core i5-8350U.

It is true that on historical dataset, SH-VSM is outperformed by STD-GST in most metrics. However, we believe that such low performance can be compensated with the introduction of two additional hints for suspecting plagiarism and collusion. Style similarity can be used to capture students with obvious attempts of plagiarism or collusion. Dishonesty probability can be used to either differentiate non-coincidental from coincidental similarity, or identify who are the culprits and what kind of dishonesty they do.

5. CONCLUSIONS AND FUTURE WORK

This paper proposes a similarity detection for source code plagiarism and collusion in academia via style and dishonesty probabilities, derived from the programming style. The technique is more effective than a common technique in academia for capturing non-semantic-preserving information, increasing precision with a trade-off in recall if applied on novice student code files. The technique is also able to capture student programming style with different distinguishing features, which provides higher effectiveness of our technique.

For future work, we plan to evaluate the impact of disauthorship probability (a part of dishonesty probability) in differentiating the culprits and the victims. As suggested by [37], they should be educated differently; the former should learn academic integrity while the latter should learn how to secure their files from being copied (including how to reject a request for their code). We also plan to propose a detailed approach to educate both culprits and victims, complementing existing approaches for maintaining academic integrity [5].

6. ACKNOWLEDGMENT

The authors would like to thank Sendy Ferdian from Maranatha Christian University, Indonesia for providing some parts of the introductory dataset.

7. REFERENCES

[1] M. Joy, G. Cosma, J. Y.-K. Yau, and J. Sinclair, "Source code plagiarism – a student perspective," *IEEE Transactions on Education*, vol. 54, no. 1, pp. 125–132, Feb. 2011.

[2] R. Fraser, "Collaboration, collusion and plagiarism in computer science coursework," *Informatics in Education*, vol. 13, no. 2, pp. 179–195, Sep. 2014.

[3] T. Lancaster, "Academic integrity for computer science instructors," in *Higher Education Computer Science*, Cham: Springer International Publishing, 2018, pp. 59–71.

[4] M. Devlin, "Policy, Preparation, and Prevention: Proactive minimization of student plagiarism," *Journal of Higher Education Policy and Management*, vol. 28, no. 1, pp. 45–58, Mar. 2006.

[5] J. Sheard, Simon, M. Butler, K. Falkner, M. Morgan, and A. Weerasinghe, "Strategies for maintaining academic integrity in first-year computing courses," *Proceedings of the 2017 ACM Conference on Innovation and*

Technology in Computer Science Education, 2017, pp. 244–249.

[6] J. Sheard, A. Carbone, and M. Dick, “Determination of factors which impact on IT students’ propensity to cheat,” *Proceedings of the 5th Australasian conference on Computing education - Volume 20*, 2003, pp. 119–126.

[7] Simon *et al.*, “Negotiating the maze of academic integrity in computing education,” *Proceedings of the 2016 ITiCSE Working Group Reports*, 2016, pp. 57–80.

[8] L. Prechelt, G. Malpohl, and M. Philippsen, “Finding plagiarisms among a set of programs with JPlag,” *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[9] G. Cosma and M. Joy, “An approach to source-code plagiarism detection and investigation using Latent Semantic Analysis,” *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 379–394, Mar. 2012.

[10] F.-P. Yang, H. C. Jiau, and K.-F. Ssu, “Beyond plagiarism: an active learning method to analyze causes behind code-similarity,” *Computers and Education*, vol. 70, pp. 161–172, Jan. 2014.

[11] U. Inoue and S. Wada, “Detecting plagiarisms in elementary programming courses,” *Proceedings of the 9th International Conference on Fuzzy Systems and Knowledge Discovery*, 2012, pp. 2308–2312.

[12] L. Sulistiani and O. Karnalim, “ES-Plag: efficient and sensitive source code plagiarism detection tool for academic environment,” *Computer Applications in Engineering Education*, vol. 27, no. 1, pp. 166–182, 2019.

[13] K. J. Ottenstein, “An algorithmic approach to the detection and prevention of plagiarism,” *ACM SIGCSE Bulletin*, vol. 8, no. 4, pp. 30–41, Dec. 1976.

[14] A. Parker and J. O. Hamblen, “Computer algorithms for plagiarism detection,” *IEEE Transactions on Education*, vol. 32, no. 2, pp. 94–99, 1989.

[15] M. J. Wise, “Yap3: improved detection of similarities in computer program and other texts,” *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, 1996, vol. 28, no. 1, pp. 130–134.

[16] A. M. Bejarano, L. E. García, and E. E. Zurek, “Detection of source code similitude in academic environments,” *Computer Applications in Engineering Education*, vol. 23, no. 1, pp. 13–22, Jan. 2015.

[17] J.-H. Ji, G. Woo, and H.-G. Cho, “A source code linearization technique for detecting plagiarized programs,” *Proceedings of the 12th Annual ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2007, pp. 73-77.

[18] J.-S. Lim, J.-H. Ji, H.-G. Cho, and G. Woo, “Plagiarism detection among source codes using adaptive local alignment of keywords,” *Proceedings of the 5th International Confernece on Ubiquitous Information Management and Communication*, 2011, p. 24.

[19] J.-H. Ji, G. Woo, and H.-G. Cho, “A plagiarism detection technique for Java program using bytecode analysis,” *Proceedings of the 3rd International Conference on Convergence and Hybrid Information Technology*, 2008, pp. 1092–1098.

[20] O. Karnalim, “Detecting source code plagiarism on introductory programming course assignments using a bytecode approach,” *Proceedings of the 10th International Conference on Information & Communication Technology and Systems*, 2016, pp. 63–68.

[21] O. Karnalim, “A low-level structure-based approach for detecting source code plagiarism,” *IAENG International Journal of Computer Science*, vol. 44, no. 4, pp. 501–522, 2017.

[22] W. B. Croft, D. Metzler, and T. Strohman, *Search Engines : Information Retrieval in Practice*. Addison-Wesley, 2010.

[23] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso, “Cross-language source code re-use detection using Latent Semantic Analysis,” *Journal of Universal Computer Science*, vol. 21, no. 13, pp. 1708–1725, 2015.

[24] F. Ullah, J. Wang, M. Farhan, S. Jabbar, Z. Wu, and S. Khalid, “Plagiarism detection in students’ programming assignments based on semantics: multimedia e-learning based smart assessment methodology,” *Multimedia Tools and Applications*, pp. 1-18, Mar. 2018.

[25] C. Arwin and S. M. M. Tahaghoghi, “Plagiarism detection across programming languages,” *Proceedings of the 29th Australasian Computer Science Conference*, 2006, pp. 277-286.

[26] M. Mozgovoy, S. Karakovskiy, and V. Klyuev, “Fast and reliable plagiarism detection system,” *Proceedings of the 37th Annual Frontiers in Education Conference*, 2007, pp. 11–14.

[27] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, “Efficient plagiarism detection for large code repositories,” *Software: Practice and Experience*, vol. 37, no. 2, pp. 151–175, 2007.

[28] D. Fu, Y. Xu, H. Yu, and B. Yang, “WASTK: a weighted abstract syntax tree kernel method for source code plagiarism detection,” *Scientific Programming*, vol. 2017, pp. 1–8, Feb. 2017.

[29] H.-J. Song, S.-B. Park, and S. Y. Park,

"Computation of program source code similarity by composition of parse tree and call graph," *Mathematical Problems in Engineering*, vol. 2015, pp. 1–12, Apr. 2015.

[30] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 872-881.

[31] H. Kikuchi, T. Goto, M. Wakatsuki, and T. Nishino, "A source code plagiarism detecting method using alignment with abstract syntax tree elements," *Proceedings of the 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2014, pp. 1–6.

[32] L. Wang, L. Jiang, and G. Qin, "A search of verilog code plagiarism detection method," *Proceedings of the 13th International Conference on Computer Science & Education*, 2018, pp. 1–5.

[33] J.-Y. Kuo, H.-K. Cheng, and P.-F. Wang, "Program plagiarism detection with dynamic structure," *Proceedings of the 7th International Symposium on Next Generation Electronics*, 2018, pp. 1–3.

[34] M. Joy and M. Luck, "Plagiarism in programming assignments," *IEEE Transactions on Education*, vol. 42, no. 2, pp. 129–133, 1999.

[35] J. Petrik, D. Chuda, and B. Steinmüller, "Source code plagiarism detection: the Unix way," *Proceedings of the 15th International Symposium on Applied Machine Intelligence and Informatics*, 2017, pp. 467–472.

[36] M. El Bachir Menai and N. S. Al-Hassoun, "Similarity detection in Java programming assignments," *Proceedings of the 5th International Conference on Computer Science & Education*, 2010, pp. 356–361.

[37] A. E. Budiman and O. Karnalim, "Automated hints generation for investigating source code plagiarism and identifying the culprits on in-class individual programming assessment," *Computers*, vol. 8, no. 1, p. 11, Feb. 2019.

[38] P. Vamplew and J. Dermoudy, "An anti-plagiarism editor for software development courses," *Proceedings of the 7th Australasian Conference on Computing Education*, 2010, pp. 83–90.

[39] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[40] O. Karnalim and R. Mandala, "Java Archives Search Engine using Byte Code as Information Source," *Proceedings of the 2014 International Conference on Data and Software Engineering (ICODSE)*, 2014, pp. 1–6.

[41] T. Mitchell, *Machine Learning*. McGraw-Hill Education, 1997.

[42] I. H. Witten, E. Frank, and M. A. Hall, *Data mining : practical machine learning tools and techniques*. Morgan Kaufmann, 2011.

[43] Y. D. Liang, *Introduction to Java programming, comprehensive version (9th Edition)*. Pearson, 2013.

***Oscar Karnalim**, graduated with a Bachelor of Engineering degree from Parahyangan Catholic University in 2011, and completed his Master degree at Bandung Institute of Technology (ITB) in 2014. His interest is about computer science education, focusing on source code plagiarism and collusion. He works at Maranatha Christian University as a full-time lecturer. Currently, he is pursuing a PhD in Information Technology at University of Newcastle, Australia.*

***Gisela Kurniawati**, graduated with a Bachelor of Computer degree from Maranatha Christian University in 2019. Her interest is about computer science education, focusing on student behaviours and educational tools. She works at Maranatha Christian University as a full-time lecturer. She is also a software developer and designer on various freelance projects.*