Informatics in Education, 2019, Vol. 18, No. 2, 321–344

DOI: 10.15388/infedu.2019.15

Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation

Oscar KARNALIM[1], Setia BUDI[1], Hapnes TOBA[1], Mike JOY[2]
[1]*Faculty of Information Technology, Maranatha Christian University, Bandung, Indonesia*
[2]*Department of Computer Science, University of Warwick, Coventry, United Kingdom*
e-mail: {oscar.karnalim, setia.budi, hapnestoba}@it.maranatha.edu, m.s.joy@warwick.ac.uk

Received: March 2019

Abstract. Source code plagiarism is an emerging issue in computer science education. As a result, a number of techniques have been proposed to handle this issue. However, comparing these techniques may be challenging, since they are evaluated with their own private dataset(s). This paper contributes in providing a public dataset for comparing these techniques. Specifically, the dataset is designed for evaluation with an Information Retrieval (IR) perspective. The dataset consists of 467 source code files, covering seven introductory programming assessment tasks. Unique to this dataset, both intention to plagiarise and advanced plagiarism attacks are considered in its construction. The dataset's characteristics were observed by comparing three IR-based detection techniques, and it is clear that most IR-based techniques are less effective than a baseline technique which relies on Running-Karp-Rabin Greedy-String-Tiling, even though some of them are far more time-efficient.

Keywords: source code plagiarism, dataset, programming, computer science education.

1. Introduction

Source code plagiarism is an act of reusing other people's code with no (or improper) acknowledgement toward the original works (Cosma & Joy, 2008). It is an emerging issue in Computer Science (CS) education (Simon *et al.*, 2018); as grades may fail to reflect the students' real capabilities. Such plagiarism detection is not an easy task since the number of investigated source code files is typically high for three reasons: programming assessments are often given at regular intervals, such as weekly (Kustanto & Liem, 2009); for each task in one assessment, the source code files should be compared to each other (Moussiades & Vakali, 2005); some students could be tempted to plagiarise even on easy tasks, since plagiarising source code is easy (Rab-

bani & Karnalim, 2017). In order to support CS educators on detecting the issue, they should be assisted with an automated detection tool such as JPlag (Prechelt, Malpohl, & Philippsen, 2002).

In terms of a detection mechanism, a number of techniques have been proposed (Karnalim, 2017). Some of them focus more on effectiveness factors (such as accuracy and the capability to detect complex modification) while the others focus on the efficiency (such as processing time). Nevertheless, they can be challenging to compare with each other, since most techniques are evaluated using their own dataset and the dataset is not publicly accessible. It is true that some datasets can be obtained by asking the corresponding author directly. However, such a mechanism can take a considerable amount of time – the author may not be able to reply promptly since they could be busy or taking a short leave. Further, the datasets (that are stored on the corresponding author's local repository) may be missing or corrupted due to technical problems.

Another important issue related to existing datasets is that some of them may not represent real plagiarism cases. These datasets may be formed with no intention of plagiarising. Further, they may only focus on trivial plagiarism attacks; occasionally, it is assumed that plagiarists to have poor academic performance – even though this is not always true (Rabbani & Karnalim, 2017).

Most detection techniques are evaluated by perceiving plagiarism detection as a searching task or IR task – where similarity degree acts as a plagiarism measurement. The original code acts as a query while its plagiarised code files act as relevant documents and its non-plagiarised code files act as irrelevant documents. The plagiarised and non-plagiarised code files are ranked based on their degree of similarity toward the original, and a detection technique is considered as effective if all plagiarised code files are highly ranked.

This paper presents a dataset for source code plagiarism detection. Specifically, the dataset is modeled for evaluation from an IR perspective. Unique to this dataset, plagiarised code files are formed with the intention of plagiarising – plagiarists are explicitly instructed to plagiarise. Further, these code files contain advanced plagiarism attacks in addition to the trivial ones, since the plagiarists are instructed to plagiarise according to plagiarism levels defined by Faidhi & Robinson (1987), where some of these levels are the advanced ones.

The dataset characteristics were observed by comparing detection techniques derived from three popular IR retrieval models: the Vector Space Model, Latent Semantic Indexing, and the Language Model (Croft, Metzler, & Strohman, 2010).

2. A Review of Automated Source Code Plagiarism Detection

Research on automated source code plagiarism detection has been growing since 1970s in which the current trend has been comprehensively described by Novak, Joy, & Kermek (2019) and Novak (2016). The research covers not only the development of such detection (Prechelt *et al.*, 2002) but also comparative studies (Ahadi & Mathieson, 2019), dataset suitability for detection analysis (Mirza, Joy, & Cosma, 2017), experiences of ap-

plying the detection in the education process (Pawelczak, 2018), and preprocessing for higher accuracy in a particular condition (Sun *et al.*, 2019).

Most automated source code plagiarism detection typically works in two consecutive phases: tokenization and comparison. Technique classifications for both phases can be seen on Fig. 1.

Tokenization converts source code files to an intermediate representation prior to comparison. Among available representations, the source code token sequence is the most common one, and is a sequence of structure-preserving terms found in the code files. It has been adopted in a number of studies (Sulistiani & Karnalim, 2019) in which some have been matured as tools – e.g., JPlag (Prechelt *et al.*, 2002) and YAP (Wise, 1996). It has been argued that the source code token sequence may not be comprehensive enough for representing the code files, and hence more-advanced representations have been introduced. Other techniques which have been proposed include: the use of abstract syntax trees containing source code tokens in their syntactic structure (Fu *et al.*, 2017; Ganguly *et al.*, 2018); program dependency graphs which describe how each instruction relies on other instructions (Liu *et al.*, 2006); and low-level token sequences extracted from the binary files of compiled source code (Karnalim, 2017, 2019; Rabbani & Karnalim, 2017).

The comparison phase measures the similarity degree between two source code files based on their intermediate representation. This phase can be categorised further to three techniques from how they define similarity degrees (Al-Khanjari *et al.*, 2010; Karnalim, 2017), namely structure-based, attribute-based, and hybrid.

A structure-based technique determines similarity based on the structure of intermediate representation. If either a source code or low-level token sequence is used, its similarity degree is commonly measured with a string-matching algorithm – e.g., Running-Karp-Rabin Greedy-String-Tiling (RKRGST) (Wise, 1996) or Local Alignment (Smith & Waterman, 1981). Otherwise, domain-specific similarity measurement could be used. Two examples of such kind of measurement are a tree kernel algorithm (Fu *et al.*, 2017) and graph isomorphism measurement (Liu *et al.*, 2006).

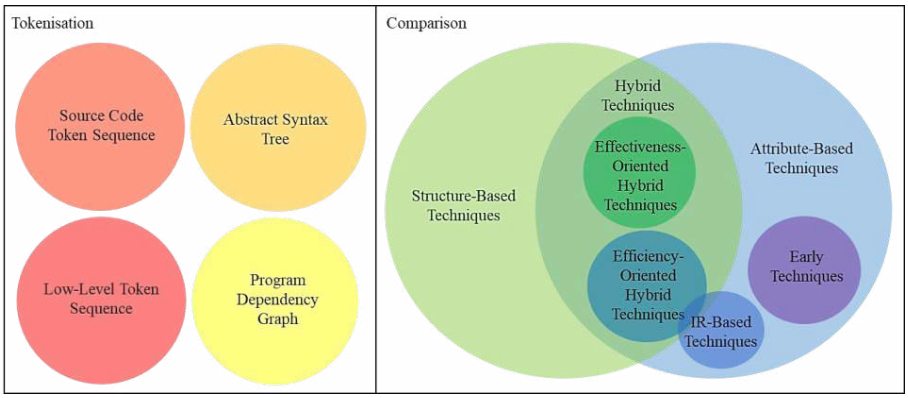

Fig. 1. A Venn diagram classifying source code plagiarism detection techniques based on tokenization and comparison phases. In general, the comparison phase has more classification variants.

A structure-based technique is usually considerably time-inefficient (Sulistiani & Karnalim, 2019). Most of its similarity measurements take at least quadratic time complexity while, for each assessment, its number of comparisons is typically high. Therefore, an attribute-based technique is introduced, which calculates similarity based on a set of attributes obtained by breaking down the structure of the intermediate representation. In this way, the similarity degree can be determined in a short time since the structure order is not considered as a large sequence. For example, techniques proposed by Moussiades & Vakali (2005) and Ohmann & Rahal (2015) utilise clustering techniques that rely on token mnemonics and their occurrences as vectors. Another example is a technique which calculates shared information between tokens with Kolmogorov complexity (Chen *et al.*, 2004).

Early detection techniques were generally attribute-based ones, where similarity between two source code files are defined based on the occurrence frequencies of several programming aspects. A technique proposed by Ottenstein (1976) considered the number of operators, operands, distinct operators, and distinct operands as its attributes for detection. Other early techniques (Donaldson, Lancaster, & Sposato, 1981; Faidhi & Robinson, 1987; Grier, 1981) expanded those attributes for higher accuracy.

IR is a mechanism to extract important information from a set of documents (Croft *et al.*, 2010). It is commonly applied on search engines such as Google and Bing to provide relevant information from numerous documents in a short period of time. In the context of attribute-based techniques, this mechanism is used to either detect plagiarised source code files (Ganguly *et al.*, 2018; Karnalim, 2019), compare source code files based on their semantics (Ullah *et al.*, 2018), or group source code files with similar semantics (Cosma & Joy, 2012a, 2012b).

A hybrid technique combines the structure-based and attribute-based techniques, which usually aims for either higher effectiveness or efficiency. In this paper, the former is called an effectiveness-oriented hybrid technique while the latter is called efficiency-oriented hybrid technique.

An effectiveness-oriented hybrid technique is developed under an argument that structure-based and attribute-based techniques are resistant to different plagiarism cases.

A technique proposed by El Bachir Menai & Al-Hassoun (2010), for instance, allows an examiner (a lecturer or teaching assistant) to see the results of both techniques; it is argued that a structure-based technique is suitable to handle cases with complex modification while an attribute-based one is suitable to handle cases with trivial modification. Another alternative to exploit both resistibilities at once is to consider the result of a particular technique as a part of the input to another. Techniques proposed by Engels, Lakshmanan, & Craig (2007) and Poon *et al.* (2012) utilise the result of a string-matching technique (structure-based technique) as a feature for an attribute-based technique – which are learning- and clustering-based techniques respectively. A technique proposed by Kuo, Cheng, & Wang (2018) measures the similarity from three aspects (variable, comment, and function) using Cosine correlation; in which the IDE-generated comments are excluded from comparison via a string-matching technique.

Efficiency-oriented hybrid techniques have been developed because a structure-based technique can be more effective than an attribute-based technique (Verco & Wise, 1996), even though the latter can be more time-efficient (Burrows, Tahaghoghi, & Zobel, 2007; Sulistiani & Karnalim, 2019). Such a technique utilises an attribute-based technique (with IR-based measurement) as an initial filter for its structure-based technique, which is based on string-matching algorithm (Burrows, Tahaghoghi, & Zobel, 2007); only source code pairs for which the attribute-based similarity degree is higher or equal to a particular threshold are passed to the structure-based one. This technique has been adapted by Sulistiani & Karnalim (2019), along with new similarity degree normalisation mechanisms that are sensitive to sub-sequence rearrangement.

Some works extend existing techniques instead of directly proposing new ones, such as a meta-tool which combines several source code plagiarism detection techniques (Portillo-Dominguez *et al.*, 2017). A technique requiring steps for detecting (and investigating) source code plagiarism is expected to form guidance for examiners (Kermek & Novak, 2016). This is complemented by research which captures examiner perspectives about how to inform academic integrity in programming (Simon *et al.*, 2018). An active learning method has further been proposed to teach students that source code similarity does not always entail plagiarism and to enrich student perspectives about similarity (Yang, Jiau, & Ssu, 2014). Those perspectives could be used as references when the examiner needs to inform students about academic integrity. Works proposed by Chuda *et al.* (2012), Cosma & Joy (2008), Cosma *et al.* (2017), Joy *et al.* (2011), and Zhang *et al.* (2014) summarise examiner and student perspectives about source code plagiarism, and can be used to gain a consensus about the definition of source code plagiarism.

Even though they are high in number, these detection techniques may be challenging to compare with each other, and since most of the techniques are evaluated with their own datasets, many of these datasets are not publicly available. Further, some datasets may be formed with no intention of plagiarising and only focus on trivial plagiarism attacks. These phenomena could inhibit the growth of this kind of research, and may lead to inconsistent and limited findings.

3. The Dataset

This paper contributes in providing a dataset for source code plagiarism detection. The dataset is expected to make existing source code plagiarism detection techniques comparable to each other. Unique to this dataset, plagiarised code files are created with the intention of plagiarising and they do not only cover trivial plagiarism attacks. This section describes how the dataset is formed, and provides a general overview of the dataset.

Due to a large variety of ways for evaluating detection techniques, the dataset is specifically tailored only for evaluation from an IR perspective. The original code is a query while its plagiarised code files are the relevant documents and its non-plagiarised code files are the irrelevant ones.

3.1. *Methodology*

The dataset is formed in five stages (see Fig. 2). At the first stage, assessment tasks – which are the main references to form the dataset – are defined. These tasks are advised to cover various programming materials so that a wide range of plagiarism attacks on the dataset may take place. Once the tasks have been defined, one or more solutions per task are provided; each solution corresponds to one original code (or query).

At the second stage, contributors are selected and categorised to two groups: plagiarists and non-plagiarists. Plagiarists will create plagiarised code files (or relevant documents) per the original code while non-plagiarists will create non-plagiarised code files (or irrelevant documents) per the assessment task to which the code relates. It is advised to use teaching assistants (TAs) as the contributors, considering that they can have a lot of experience with manual source code plagiarism detection. They will be able to both include various plagiarism attacks and complete the assessment tasks. It is true that, as plagiarists, the TAs' intention to plagiarise may not be comparable to real plagiarists', since they are not concerned about being caught or needing to get a good mark. However, TAs are more cooperative than real plagiarists, since most real plagiarists do not want to admit their behaviour regardless of the degree of evidence. Further, real plagiarists' plagiarism attacks may be more limited than TAs', since their attacks only represent their own perspective while TAs' represent many different student perspectives (obtained through the TAs' experiences).

When compared to the lecturers – who could also be the researchers in charge in the research project – TAs arguably have a more objective perspective toward source code plagiarism. The TAs have no conflicts of interest with the project. In addition, they directly engage with students and the source code at the laboratory sessions.

Prior to the construction of the dataset, all contributors are required to read terms and conditions related to their contribution, to acknowledge that their code files will be published as a public dataset with their identity anonymized.

At the third stage, plagiarised and non-plagiarised code files are created. The former are created by TAs adopting the roles of plagiarists while the latter are created by TAs who do not plagiarise.

Contributors who are labeled as plagiarists are instructed to plagiarise original code files. To assure that they plagiarise instead of solving the assessment tasks by themselves, they are not allowed to see the tasks, and only original code files are accessible to them. In such a manner, their intention of plagiarising will emerge. Moreover, to involve advanced plagiarism attacks in addition to simple ones, these contributors are explicitly instructed to consider those advanced attacks while plagiarising. If they are not familiar with the advanced attacks, examples are provided.

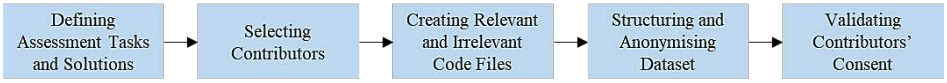

Fig. 2. How the dataset is formed. Five consecutive phases are required, starting from defining assessment tasks and solutions to validating contributors' consent.

Contributors who are labeled as non-plagiarists are instructed to solve the assessment tasks by themselves. To assure their honesty, they can only access the tasks, and the original and plagiarised code files are not accessible to them. If possible, those assistants are gathered in one room at a particular time and they should complete their tasks there without seeing others' work. Otherwise, they can be instructed to complete the tasks in their own time and place as long as they have high integrity. It is worth noting that Internet access will not pose a threat to the validity of the dataset as long as the original and plagiarised code files are not published online.

At the fourth stage, all code files are stored in a particular directory structure with their contributors' data anonymized. In terms of directory structure, each original code is paired with its respective plagiarised and non-plagiarised code files, and therefore stored under a *case* directory with three sub-directories: *original*, *plagiarised* and *non-plagiarised* (see Fig. 3). These sub-directories store the code files for which the category is similar to their name. If there is more than one code file under a particular category, each code is stored in a separate sub-directory to avoid conflicting file names. It is important to note that, considering the code files may not be uniform in terms of format and structure, all project-related files and information are removed. Upon removal, these code files are compiled once more to ensure that they still work properly.

The dataset is anonymized for privacy reasons. Getting involved on providing a plagiarism dataset may be harmful for the contributors' reputation (especially for those

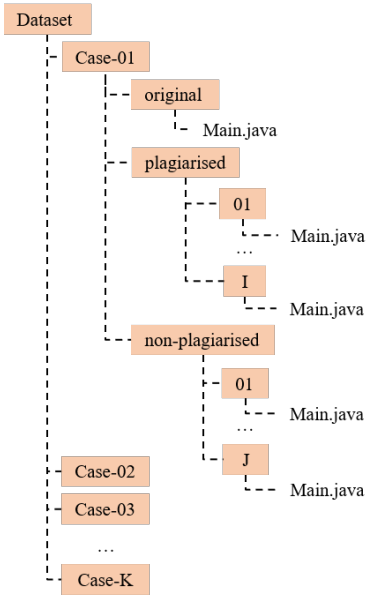

Fig. 3. A sample directory structure of the dataset where a shaded-backgrounded text refers to a directory. *I* refers to the number of plagiarised code files in case-01, *J* refers to the number of non-plagiarised code files in case-01, and *K* refers to the number of cases included in the dataset. *plagiarised* and *non-plagiarised* store their source code files under sub-directories since they have more than one source code.

who act as the plagiarists). All private information from both source code filename and content is replaced with their respective MD5 hash values (Zhong, Wan, & Kong, 2016). The replacement works in two phases. First, a simple program replaces each item of predefined information with its hash value. Afterwards, the source code files are checked manually to assure that no private information remains, and any remaining information is replaced with a hash value in a semi-manual manner.

Finally, once the dataset is ready for publication, contributors' consents are validated to make sure that they know what their code files will look like when published. Contributors' consents are validated through email, since paper-based consent forms are quite inconvenient to use and not all contributors can be met in person. Each contributor is sent an email containing a link to the dataset and restated terms and conditions. If they still agree to contribute, they can reply to the email with their name written on the replying email's body, otherwise, they can provide a reason why they have changed their decision. In this way, contributors' consents are stored on an email account with a non-modifiable timestamp. We would argue that this mechanism is quite secure and provides a trusted consent proof.

3.2. *Result*

Our plagiarism dataset (Karnalim, Budi, Toba, & Joy, 2019) consists of 467 source code files, depicting seven introductory programming assessment tasks. Each task has one original code file, 15 non-plagiarised code files, and up to 54 plagiarised code files – which are classified further according to six of the seven plagiarism levels defined by Faidhi & Robinson (1987). All code files are written in the Java programming language.

Original code files were taken from Liang (2013) where their task details were created by the first author of this paper through an observation about how these code files work. The details are (for conciseness, sample input and output for each task are not listed):

- T1 (Output): Write a program that prints "Welcome to Java" five times.
- T2 (Input): Write a program that accepts the radius & length of a cylinder and prints the area & volume of that cylinder. All inputs and outputs are real numbers.
  Cylinder's area = radius ∗ radius ∗ 3.14159
  Cylinder's volume = area ∗ length.
- T3 (Branching): Write a program that accepts the weight (as a real number representing pound) and height (as two real numbers representing feet and inches respectively) of a person. Upon accepting input, the program will show that person's BMI (real number) and a piece of information stating whether the BMI is categorised as underweight, normal, overweight, or obese.
  A person is underweight if BMI < 18.5; normal if 18.5 ≤ BMI < 25; overweight if 25 ≤ BMI < 35; or obese if BMI ≥ 35.
  Height = feet ∗ 12 + inches

BMI = weight ∗ 0.45359237 / (height ∗ 0.0254)²

- T4 (Looping): Write a program that shows a conversion table from miles to kilometers where one mile is equivalent to 1.609 kilometers. The table should display the first ten positive numbers as miles and pair them with their respective kilometer representation.
- T5 (Method): Write a program that accepts an integer and displays that integer with its digits shown in reverse. You should create and use a method *void reverse(int number)* which will show the reversed-digit form of the parameterised number.
- T6 (Array): Write a program that accepts 10 integers and shows them in reversed order.
- T7 (Matrix): Write a program that accepts a 4 x 4 matrix of real numbers and prints the total of all numbers placed on the leading diagonal of the matrix. You should create and use a method *double sumMajorDiagonal(double[][] m)* which will return the total of all numbers placed on the leading diagonal of the parameterised matrix.

The plagiarised code files were created by assigning nine teaching assistants as plagiarists. These assistants were proficient in Java programming and had sufficient experience with manual source code plagiarism detection. For each original code, these assistants were instructed to create six plagiarised code files based on the plagiarism level taxonomy defined by Faidhi & Robinson (1987) – excluding the lowest level since it leads to no modifications between original and plagiarised code; one plagiarised code corresponds to one plagiarism level. The plagiarised code files have been used to compare the effectiveness of two low-level techniques by Karnalim & Budi (2018). However, these code files have not been published.

Plagiarism levels defined in Faidhi & Robinson (1987) map source code plagiarism attacks to seven categories based on their difficulty. Fig. 4 shows how these categories interact with each other and their signature attacks as phrased by Karnalim & Budi (2018). The easiest (or the most trivial) one is level-0 which is verbatim copy (no modifications exist between original and plagiarised code). The most challenging one is level-6

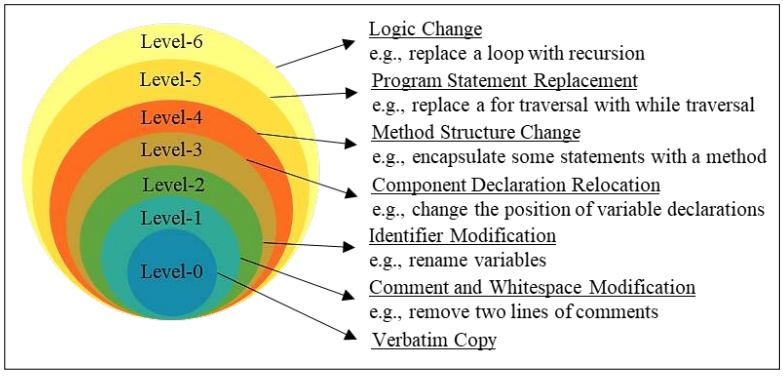

Fig. 4. Plagiarism Level Taxonomy as defined by Faidhi & Robinson (1987). It has seven levels where a particular level's attack also covers its lower levels' attacks.

which is logic change, and is only considered as a plagiarism evidence if it occurs in combination with other attack levels. It is important to note that plagiarised code files on a particular level may have plagiarism attacks from lower levels. For example, level-5 plagiarised code may contain identifier modification (which is the level-2 attack). Using this plagiarism level taxonomy, plagiarists were guaranteed to put advanced attacks in some plagiarised code files.

The number of plagiarised code files per task can be seen in Table 1. Since there are nine plagiarists, it is expected that each level per task has nine plagiarised code files. However, some of them have fewer code files, since code files which excessively violate their targeted plagiarism level (e.g., a level-1 code that utilises a level-6 attack) are excluded.

The non-plagiarised code files were created with the help of 15 teaching assistants (as the non-plagiarists) who were instructed to solve the assessment tasks without seeing other code files. Since they were trusted in terms of integrity, we let them create the non-plagiarised code files at home. Although we can use any code files for the non-plagiarised ones, we would argue that these code files should be created by solving similar assessment tasks as the original and plagiarised code files; they would be able to share coincidental similarity with other code files, making the dataset more realistic.

Original, plagiarised, and non-plagiarised code files were then stored and anonymized. In our case, plagiarised code files were grouped further under their plagiarism level (where each level refers to one sub-directory of *plagiarised* directory). Moreover, removed private information covers student names, student academic IDs, laptop product, and social media account names.

All participants completed the consent process and agreed to publish their code files for non-profit research purposes.

The statistics from our dataset can be seen in Table 2, which has 467 source code files with 59,201 tokens in total, and each code has an average of 126 tokens.

Three important remarks should be made about the dataset. First, the plagiarism level taxonomy provided by Faidhi & Robinson (1987) may not be strictly followed due to different interpretations by the authors and the plagiarists of the taxonomy. However, we can guarantee that the plagiarism attacks become more advanced as the plagiarism level is increased. Second, some code files may not solve their respective

Table 1
The number of plagiarised code files per task

Task ID	Lv1	Lv2	Lv3	Lv4	Lv5	Lv6	Total
T1	9	5	6	6	5	9	40
T2	9	9	9	9	9	9	54
T3	8	9	8	9	9	9	52
T4	9	9	9	9	9	9	54
T5	9	8	9	9	9	9	53
T6	8	8	8	9	9	9	51
T7	8	8	8	9	9	9	51

Table 2
Dataset Statistics

Metric	Value
Number of original code files	7
Number of plagiarised code files	355
Number of non-plagiarised code files	105
Total number of code files	467
Total number of tokens	59,201
Total number of distinct tokens	540
Maximum number of tokens per code	286
Minimum number of tokens per code	40
Average number of tokens per code	126

task perfectly due to the contributors' carelessness (e.g., some code files may have minor output errors), but each one compiles. These code files are still included to keep the naturalness of our dataset, and in real conditions, not all submitted code files get a perfect score. Third, the dataset may not cover all types of plagiarism attacks on introductory programming, since the tasks used to form the dataset are only a subset of introductory programming tasks and the plagiarism attacks are purely defined by the plagiarists.

4. Observing the Dataset Characteristics: A Comparative Study of IR-Based Techniques

To provide more details about the dataset's characteristics, a comparative study with such a dataset on board was performed, involving three IR-based source code plagiarism detection techniques (IR-based techniques). The findings can be a broad overview of the dataset characteristics when a particular technique is applied. Those IR-based techniques are derived from three popular retrieval model: the Vector Space Model (VSM), Latent Semantic Indexing (LSI), and the Language Model (LM) (Croft *et al.*, 2010); they are referred to as VSM-based technique (VSM-t), LSI-based technique (LSI-t), and LM-based technique (LM-t) respectively. This section explains how these techniques are compared and what the findings resulting from the study are.

4.1. *Methodology*

Fig. 5 depicts how the IR-based techniques work per the original code. First, they accept all source code files and translate them to token sequences; one source code results in one sequence. The translation is performed with the help of ANTLR (Parr, 2013) where comments and whitespace tokens are removed. An example of the translation (or tokenization) process can be seen in Fig. 6.

Fig. 5. IR-based technique works in three phases per original code. It accepts the code with their plagiarised and non-plagiarised code files as the inputs and returns plagiarism-suspected code files as the results.

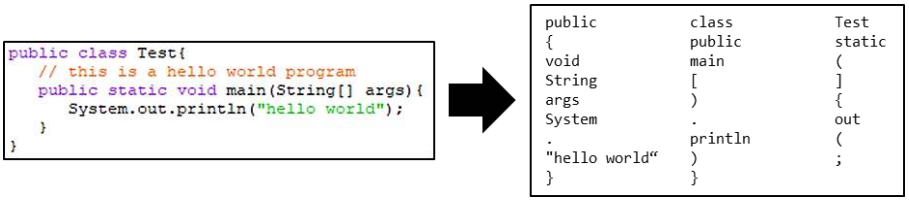

Fig. 6. An example of a translation process from a source code file to a token sequence; all comments and whitespaces are excluded from the sequence.

Second, an index for each sequence is built. Index is a data structure containing key-value pairs where key refers to a token mnemonic and value refers to the occurrence frequency of that mnemonic. To illustrate this, let us assume we have a token sequence: *{x, =, x, +, y, ;}*. The resulted index will have 5 tuples which are: *{'x':2}, {'=':1}, {'+':1}, {'y':1},* and *{';':1}*.

Third, the original code's index acts as a query to rank all plagiarised and non-plagiarised code files based on the given IR-based measurement. An IR-based technique is considered as effective if it can distinguish the plagiarised code files from the non-plagiarised ones and put them at the top of the list.

VSM, LSI, and LM (Croft *et al.*, 2010) work in different ways for determining similarity. VSM models the code files as vectors based on their token occurrence frequencies, and the similarity between two vectors is commonly calculated using cosine similarity. LSI is an improved model of VSM where the vector similarity is defined based on concept relation (generated from Singular Value Decomposition) instead of token occurrence frequency. LM considers the probability distribution over token sequences. In our case, a query-likelihood LM model is used; plagiarised and non-plagiarised code files are ranked based on the probability of given original code in their language model. Each document is scored as in (1) where Q refers to the original code, *q* refers to one original code's token, *D* refers to a plagiarised or non-plagiarised code, *C* refers to the whole collection, and *f(a,B)* returns the occurrence frequency of *a* in *B*. It is important to note that the two constants in (1) – i.e., 0.3 and 0.7 – are selected due to their success on previous LM-related research (Croft *et al.*, 2010).

$$s(Q,D,C) = \sum_{q \in Q} \log\left(\frac{0.3 * f(q,D)}{|D|} + \frac{0.7 * f(q,C)}{|C|}\right) \quad (1)$$

String matching based detection technique with RKRGST (Wise, 1996) is included in our study as a baseline, since it is commonly used for source code plagiarism detec-

tion (Sulistiani & Karnalim, 2019). This technique (which is labeled as RKRGST-t) works in a similar manner as IR-based techniques except that it requires no indexes and the similarity is based on RKRGST toward given source code files' token sequences. In our context, RKRGST's similarity degree is calculated with two as its minimum matching length and average normalisation (Prechelt *et al.*, 2002; Sulistiani & Karnalim, 2019) as its normalisation mechanism. Average normalisation is performed as in (2) where $C1$ and $C2$ are given source code files' token sequences; the number of matched tokens – labeled as $match(C1,C2)$ – is doubled and divided by the total number of both code files' tokens.

$$norm(C1, C2) = \frac{2 * match(C1, C2)}{|C1| + |C2|} \tag{2}$$

In order to provide more comprehensive analyses, n-gram and Term-Frequency-Inverse-Document-Frequency (TF-IDF) weightings (Croft *et al.*, 2010) are also considered in this study. N-gram is a mechanism which considers *n* contiguous tokens as one token, where higher *n* usually leads to more contextual features. For IR-based techniques, this mechanism is applied right after the translation (or tokenisation) process. TF-IDF weighting is a mechanism to make unique tokens become more important than the common ones, and on most occasions, this technique results in higher effectiveness. It is applied in IR-based techniques immediately before the indexes are built.

Fig. 7 depicts how the comparative study is performed. Two evaluation metrics are considered: Mean Average Precision (MAP) – for effectiveness – and time complexity – for efficiency. As seen in (3), MAP (Croft *et al.*, 2010) is a rank-sensitive proportion between the number of plagiarised code files and the total number of plagiarised and non-plagiarised code files (D), averaged for all original code files (O). Average precision is calculated as in (4) where *prec(i)* is precision at position *i*, *is_rel(i)* returns true if the i^{th}-ranked code was plagiarised from the original code (o), and *relevant(o)* refers to the original's plagiarised code files. Precision is calculated as in (5) where *relevant_retrieved(o,i)* refers to the original's plagiarised code files which rank is higher or equal to *i*. Time complexity is an algorithmic approach to calculate the proportion between given input and processing time (Levitin, 2012). It is used to replace real execution time for measuring time efficiency since it is unaffected by hardware and operating system dependencies on our dataset (which is limited in size). Considering that all techniques' numbers of comparisons are the same in this context, time complexity is only measured per comparison.

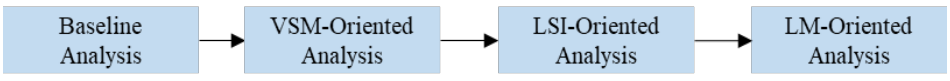

Fig. 7. Four comparative study phases that are conducted in order. It starts with analysing the characteristics of the baseline technique and ends with analysing the characteristics of LM-based techniques.

$$MAP(O,D) = \frac{\sum_{o \in O} avg\_prec(o,D)}{|O|} \tag{3}$$

$$avg_{prec(o,D)} = \frac{\sum_{i=1}^{|D|} prec(o,i) * is_{rel(i)}}{|relevant(o)|} \tag{4}$$

$$prec(o,i) = \frac{|relevant\_retrieved(o,i)|}{i} \tag{5}$$

Baseline analysis measures how effective the baseline technique (RKRGST-t) is from a plagiarism level perspective. The effectiveness toward a particular level is calculated by considering only plagiarised code files from that level. The result of this analysis is used as the baseline effectiveness for the remaining analyses.

VSM-oriented analysis compares VSM-based techniques with the baseline technique from a plagiarism level perspective. In this analysis, the impact of n-gram and TF-IDF weighting are also measured. The former is measured by comparing four *n*-gram scenarios, starting from unigram (n = 1) to quartogram (n = 4), for each plagiarism level. The latter is measured by comparing the absence with the presence of TF-IDF weighting toward all scenarios used for measuring the impact of *n*-gram.

LSI-oriented analysis works similar to VSM-oriented analysis except that it utilises LSI-based techniques instead of VSM-based techniques. To measure the impact of the LSI number of dimensions, this analysis is further broken down into two sub-analyses. The former uses two as the number of dimension, that leads to the most general semantic categorisation. The latter uses the total number of involved code files (original + plagiarised + non-plagiarised code files), which range from 21 to 25 for each original code per plagiarism level. This setting leads to the most specific semantic categorisation.

LM-oriented analysis also works similarly as VSM-oriented analysis. However, it replaces VSM-based techniques with LM-based techniques. Further, it is not featured with the analysis of TF-IDF weighting; LM, to some extent, works in a similar manner as TF-IDF weighting.

4.2. *Baseline Analysis*

The effectiveness of our baseline technique (RKRGST-t) is inversely proportional to the plagiarism level (see Fig. 8). Further observation shows that, at higher levels, more advanced plagiarism attacks are used and those attacks are hard to detect. Level-1 is the only level where RKRGST-t generates the perfect MAP (100%). Its signature plagiarism attacks are nullified prior to comparison, since comment and whitespace tokens are automatically removed by ANTLR. In terms of efficiency, RKRGST-t takes cubic complexity in theory and quadratic complexity in practice (Prechelt *et al.*, 2002; Wise, 1996).

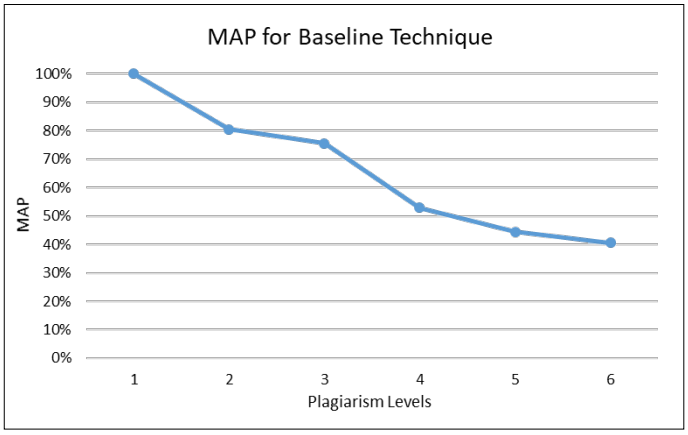

Fig. 8. MAP for RKRGST-t as the baseline technique. The horizontal axis refers to plagiarism levels while the vertical axis refers to their corresponding MAPs.

4.3. *VSM-oriented Analysis*

In the dataset, VSM relies on 540 tokens as vectors for the unigram scenario. That number gets higher as n in n-gram is increased. Bigram, trigram, and quartogram lead to 2,336, 4,697, and 6,903 vectors respectively.

Fig. 9 shows that the VSM-t is slightly more effective than the baseline technique (RKRGST-t) as long as unigram scenario (n = 1) is not used, and on average, it generates 2.04% higher MAPs. Unigram leads to a 0.03% lower MAP than RKRGST-t since it does not consider token order. The absence of such order results in similarity degrees for both plagiarised and non-plagiarised code files resembling each other when they share similar token sets to their original code.

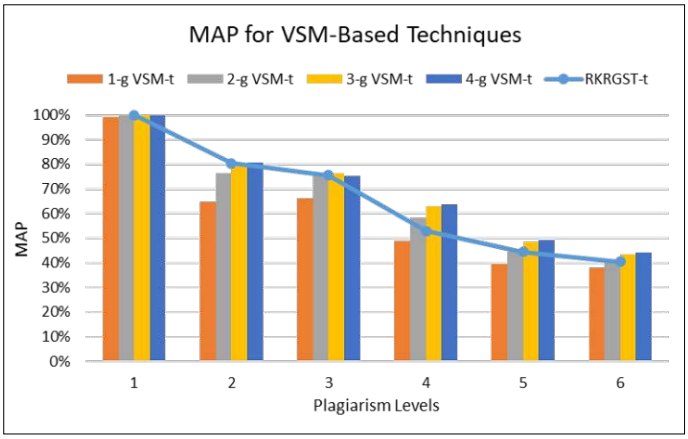

Fig. 9. MAP for VSM-based techniques. The horizontal axis refers to plagiarism levels with various n-gram settings while the vertical axis refers to their corresponding MAPs.

The token order can be introduced by increasing *n* in n-gram; each n-gram token contains *n*-ordered initial tokens. As seen in Fig. 9, *n* is proportional to MAP improvement. When *n* = 1, its average MAP improvement is -0.03% (which is a slight reduction). It then improves to 2.04%, 2.98%, and 3.17% respectively as *n* is gradually increased.

Qualitatively speaking, higher *n* provides more meaningful terms for differentiating the plagiarised from the non-plagiarised code files. To illustrate this, two unigram terms from the original code files are semicolon and dot; they are less meaningful than most quartogram (*n* = 4) terms from the same code files – e.g., *int miles = 1* and *print("Enter a 4 by 4 matrix row by row: ")*.

For the top half of the plagiarism levels, token order leads to a higher MAP when it is partially considered. VSM-t with trigram (*n* = 3) and quartogram (*n* = 4) generate higher MAPs than RKRGST-t (which are, in average, 5.58% and 6.34% higher respectively). The highest improvement occurs for the level-4 category which the most frequently applied attack is method introduction. That introduction changes the order of large token subsequences, and therefore favors techniques which rely on many short token subsequences (such as VSM-t with trigram or quartogram).

When TF-IDF weighting is applied, Fig. 10 shows that VSM-t experiences MAP reduction instead of improvement. This finding is expected since, in plagiarised code files, several identifiers are renamed. They are weighted more as mismatches since they are unique and only occur on few code files.

Theoretically, the VSM-based technique is more time-efficient than the baseline technique. It has only linear complexity (for both indexing and comparison phase) while the baseline has up to cubic time complexity.

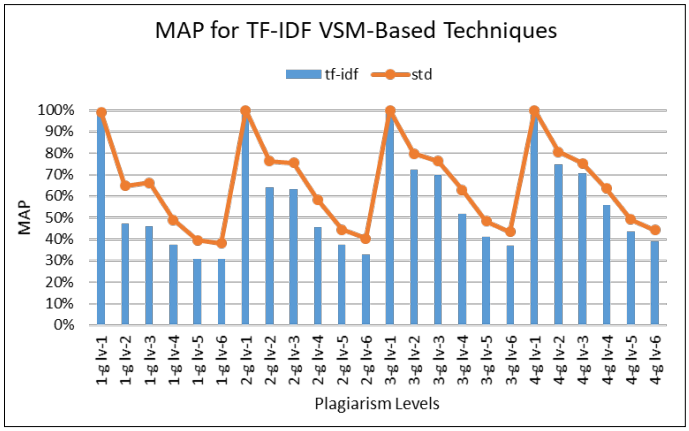

Fig. 10. MAP for TF-IDF VSM-based techniques. The horizontal axis refers to techniques involving a particular *n* in n-gram for a plagiarism level. For instance, 1-*g lv*-2 refers to a technique with unigram mechanism (*n* = 1) for level-2 plagiarism category. The vertical axis refers to these techniques' corresponding MAPs. The line corresponds to standard techniques without TF-IDF while the bars refer to those techniques with TF-IDF.

4.4. *LSI-Oriented Analysis*

Fig. 11 shows that LSI-based techniques with minimum number of dimensions (2-dim LSI-t) generate similar MAPs regardless of plagiarism levels and those MAPs are generally lower than those of the baseline (RKRGST-t). In addition, increasing the *n* in n-gram provides no clear findings. Further observation shows that both plagiarised and non-plagiarised code files generate LSI-based similarity degrees which resemble each other, adding more difficulties in separating the plagiarised code files from the non-plagiarised ones.

When TF-IDF weighting is applied, Fig. 12 depicts that it worsens 2-dim LSI-t effectiveness. However, such reduction is less significant compared to that weighting's

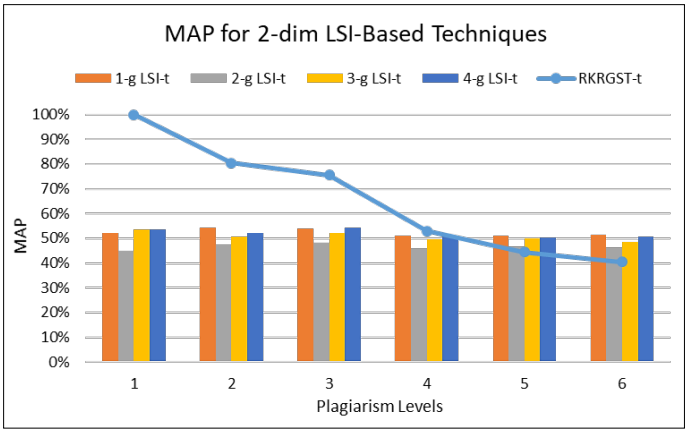

Fig. 11. MAP for 2-dim LSI-based techniques. The horizontal axis refers to plagiarism levels while the vertical axis refers to their corresponding MAPs.

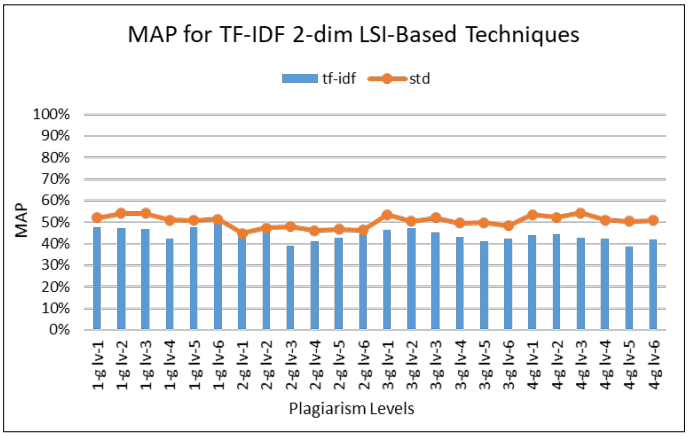

Fig. 12. MAP for TF-IDF 2-dim LSI-based techniques. The horizontal axis refers to techniques involving a particular *n* in n-gram for a plagiarism level. For instance, 1-*g lv*-2 refers to a technique with unigram mechanism ($n = 1$) for level-2 plagiarism category. The vertical axis refers to these techniques' corresponding MAPs. The line corresponds to standard techniques without TF-IDF while the bars refers to those techniques with TF-IDF.

impact on the VSM-t. In average, it only reduces MAP by 6.04% on 2-dim LSI-t while reducing by 7.66% on VSM-t. One of the possible reasons is that 2-dim LSI-t uses far fewer dimensions than VSM-t, and TF-IDF weighting affects more when many dimensions are considered.

Once the number of dimensions is increased (see Fig. 13), the LSI-t effectiveness pattern becomes more similar to the baseline's, and the MAP tends to be inversely proportional to the plagiarism level. Moreover, increasing *n* in n-gram starts to affect the effectiveness. A higher number of dimensions leads to more specific topic categorisation,

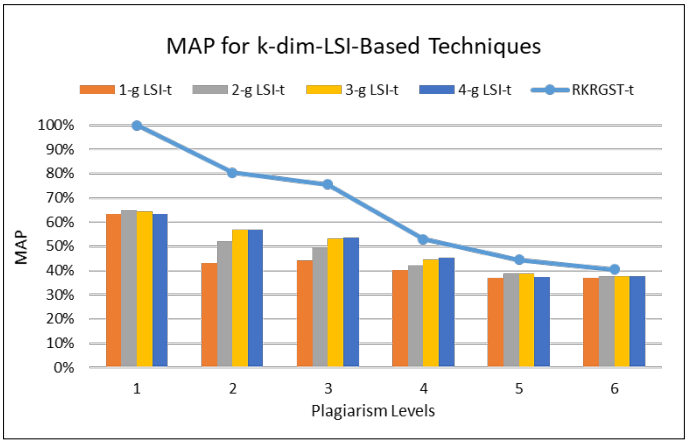

Fig. 13. MAP for k-dim LSI-based techniques. The horizontal axis refers to plagiarism levels while vertical axis refers to their corresponding MAPs.

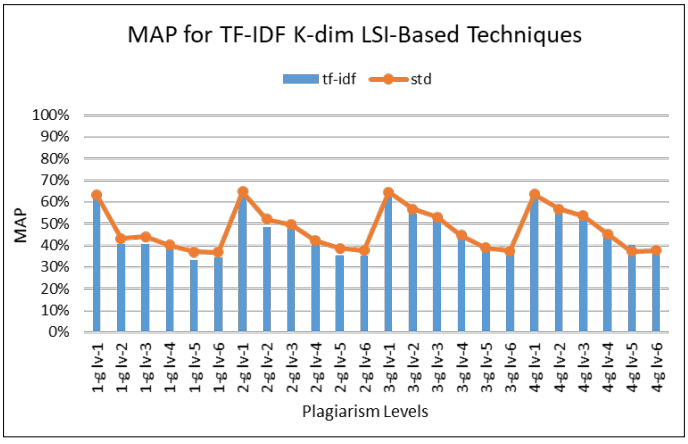

Fig. 14. MAP for TF-IDF k-dim LSI-based techniques. The horizontal axis refers to techniques involving a particular *n* in n-gram for a plagiarism level. For instance, 1-*g lv*-2 refers to a technique with unigram mechanism ($n = 1$) for the level-2 plagiarism category. The vertical axis refers to these techniques' corresponding MAPs. The line corresponds to standard techniques without TF-IDF while the bars refers to those techniques with TF-IDF.

strengthening the difference between the plagiarised and non-plagiarised code files. That topic specificity also leads to a broader similarity degree range, which is 51.39%.

The TF-IDF weighting mechanism still worsens LSI-t effectiveness when the number of dimensions is high. However, as seen in Fig. 14, its average reduction (1.13%) is less significant than its reduction on 2-dim LSI-t (6.04%).

The LSI-based technique's comparison phase is comparable to the VSM-based technique's in terms of processing time, however, its indexing phase is obviously slower. In addition to collecting token occurrence frequencies, the LSI-based technique also needs to perform Singular Value Decomposition (which commonly takes quadratic time complexity). Compared to the baseline technique, the LSI-based technique may take similar processing time, as the baseline technique also takes quadratic time complexity.

4.5. *LM-Oriented Analysis*

As presented in Fig. 15, LM-t is slightly less effective than the baseline technique (RKRGST-t) even though its pattern is fairly similar to the VSM-based technique. This finding is not surprising since LM, to some extent, works like VSM with TF-IDF weighting, and such weighting accentuates the impact of renamed identifiers (on the plagiarised code files) as mismatches.

Compared to other n-gram scenarios, unigram generates the lowest MAP on the first four plagiarism levels. Further observation shows that it generates the narrowest similarity degree range on those levels, resulting in some difficulties in differentiating the plagiarised code files from the non-plagiarised ones. Unigram's similarity degree range is only 14.54%, which is far lower than others' (21.68% for bigram, 25.32% for trigram, and 26.97% for quartogram).

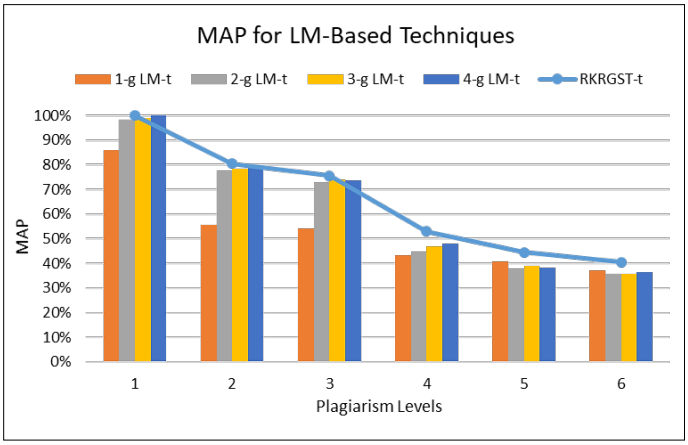

Fig. 15. MAP for LM-based techniques. The horizontal axis refers to plagiarism levels while the vertical axis refers to their corresponding MAPs.

For the last two plagiarism levels, *n*-gram does not provide a significant impact. Advanced plagiarism attacks and LM's token relation could mitigate the impact of *n*-gram in terms of exploiting shared context between original and plagiarised code files.

According to our implementation, LM-t takes more processing time than VSM-t since its measurement considers more aspects (such as local and global relation), hence it is less time efficient than VSM-t. However, it is still faster than the baseline technique since LM-t only takes linear time complexity.

4.6. *Summary*

As seen in Table 3, most IR-based techniques are less effective than the baseline technique (RKRGST-t) as they do not fully consider token order (in exchange to shorter processing time). However, this can be compensated with n-gram mechanism; higher *n* leads to more contextual information.

Among all IR-based techniques used in this study, VSM-t is the most comparable to RKRGST-t as the difference is mainly about the similarity algorithm. It can even outperform RKRGST-t when combined with n-gram mechanism.

LSI-t performs worse than VSM-t because latent relations captured from our dataset are not salient enough to differentiate plagiarised code files to the non-plagiarised ones. Despite k-dim LSI-t experiences larger reduction than 2-dim LSI-t, its result is more similar to RKRGST-t's result. The MAP is inversely proportional to the plagiarism level (see Fig. 11 and Fig. 13).

LM-t leads lower MAP as it accentuates the impact of rare tokens in which some of those tokens are a part of plagiarism attacks (e.g., identifier renaming, a plagiarism attack that replaces the names of some identifiers). This is also a reason why TF-IDF weighting seems to lower effectiveness. We presume this issue can be dealt by generalising those identifiers to the same token.

In terms of efficiency, VSM-t is more time-efficient than RKRGST-t, followed by LM-t and LSI-t respectively. This is expected as VSM-t has no need to compute additional relations like the latent ones in LSI-t.

Table 3
Averaged MAP Differences toward RKRGST-t

Metric	Unigram	Bigram	Trigram	Quartogram
VSM-t	-0.03%	2.04%	2.98%	3.17%
TF-IDF VSM-t	-17.02%	-8.48%	-3.64%	-1.63%
2-dim LSI-t	-13.36%	-19.1%	-15.03%	-13.65%
TF-IDF 2-dim LSI-t	-18.73%	-21.75%	-21.44%	-23.41%
k-dim LSI-t	-21.54%	-18.16%	-16.42%	-16.65%
TF-IDF k-dim LSI-t	-24.07%	-20.02%	-17.38%	-15.86%
LM-t	-12.81%	-4.39%	-3.5%	-2.94%

5. Conclusion

This paper proposes a dataset for evaluating source code plagiarism detection techniques from an IR perspective, where the plagiarised cases are formed with the intention of plagiarising and contain advanced plagiarism attacks in addition to simple ones. In total, there are 467 source code files depicting seven introductory programming assessment tasks. Each task has one original code, 15 non-plagiarised code files, and up to 54 plagiarised code files.

The dataset characteristics have been observed through a comparison of three IR-based detection techniques, derived from the Vector Space Model, Latent Semantic Indexing, and the Language Model. The comparison yields four main results. Firstly, in terms of effectiveness, IR-based technique with the Vector Space Model and the baseline technique derived from Running-Karp-Rabin Greedy-String-Tiling are in favour of the dataset. Secondly, in the dataset, the TF-IDF weighting mechanism does not enhance the effectiveness of the IR-based techniques, mostly due to identifier renaming. Thirdly, higher n in n-gram usually leads to more context consideration and then higher effectiveness. Finally, all IR-based techniques, except those which rely on Latent Semantic Indexing, are theoretically faster than RKRGST-based techniques.

For future work, we plan to expand the scope of the dataset with advanced programming topics (e.g., searching and sorting) and code files taken from other programming courses (e.g., Object-Oriented Programming or Algorithms and Data Structures). This expansion may enable unique and uncovered plagiarism attacks.

Acknowledgments

The authors would like to thank all the contributors who are involved in the process of dataset creation.

References

Ahadi, A., & Mathieson, L. (2019). A comparison of three popular source code similarity tools for detecting student plagiarism. *21st Australasian Computing Education Conference*, 112–117. https://doi.org/10.1145/3286960.3286974

Al-Khanjari, Z. A., Fiaidhi, J. A., Al-Hinai, R. A., & Kutti, N. S. (2010). PlagDetect: a Java programming plagiarism detection tool. *ACM Inroads*, *1*(4), 66–71. https://doi.org/10.1145/1869746.1869766

Burrows, S., Tahaghoghi, S. M. M., & Zobel, J. (2007). Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, *37*(2), 151–175. https://doi.org/10.1002/spe.750

Chen, X., Francia, B., Li, M., McKinnon, B., & Seker, A. (2004). Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, *50*(7), 1545–1551. https://doi.org/10.1109/TIT.2004.830793

Chuda, D., Navrat, P., Kovacova, B., & Humay, P. (2012). The Issue of (software) plagiarism: a student view. *IEEE Transactions on Education*, *55*(1), 22–28. https://doi.org/10.1109/TE.2011.2112768

Cosma, G., & Joy, M. (2008). Towards a definition of source-code plagiarism. *IEEE Transactions on Education*, *51*(2), 195–200. https://doi.org/10.1109/TE.2007.906776

Cosma, G., & Joy, M. (2012a). An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Transactions on Computers*, *61*(3), 379–394. https://doi.org/10.1109/TC.2011.223

Cosma, G., & Joy, M. (2012b). Evaluating the performance of LSA for source code plagiarism detection. *Informatica*, *36*(4), 409–424.

Cosma, G., Joy, M., Sinclair, J., Andreou, M., Zhang, D., Cook, B., & Boyatt, R. (2017). Perceptual comparison of source-code plagiarism within students from UK, China, and South Cyprus higher education institutions. *ACM Transactions on Computing Education*, *17*(2). https://doi.org/10.1145/3059871

Croft, W. B., Metzler, D., & Strohman, T. (2010). *Search Engines : Information Retrieval in Practice*. Addison-Wesley.

Donaldson, J. L., Lancaster, A.-M., & Sposato, P. H. (1981). A plagiarism detection system. *12th SIGCSE Technical Symposium on Computer Science Education*, *13*(1), 21–25. https://doi.org/10.1145/800037.800955

El Bachir Menai, M., & Al-Hassoun, N. S. (2010). Similarity detection in Java programming assignments. *Fifth International Conference on Computer Science & Education*, 356–361. https://doi.org/10.1109/ICCSE.2010.5593613

Engels, S., Lakshmanan, V., & Craig, M. (2007). Plagiarism detection using feature-based neural networks. *38th SIGCSE Technical Symposium on Computer Science Education*, *39*(1), 34. https://doi.org/10.1145/1227504.1227324

Faidhi, J. A. W., & Robinson, S. K. (1987). An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, *11*(1), 11–19. https://doi.org/10.1016/0360-1315(87)90042-X

Fu, D., Xu, Y., Yu, H., & Yang, B. (2017). WASTK: a weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming*, *2017*, 1–8. https://doi.org/10.1155/2017/7809047

Ganguly, D., Jones, G. J. F., Ramírez-de-la-Cruz, A., Ramírez-de-la-Rosa, G., & Villatoro-Tello, E. (2018). Retrieving and classifying instances of source code plagiarism. *Information Retrieval Journal*, *21*(1), 1–23. https://doi.org/10.1007/s10791-017-9313-y

Grier, S. (1981). A tool that detects plagiarism in Pascal programs. *12th SIGCSE Technical Symposium on Computer Science Education*, *13*(1), 15–20. https://doi.org/10.1145/800037.800954

Joy, M., Cosma, G., Yau, J. Y.-K., & Sinclair, J. (2011). Source code plagiarism–a student perspective. *IEEE Transactions on Education*, *54*(1), 125–132. https://doi.org/10.1109/TE.2010.2046664

Karnalim, O. (2017). A low-level structure-based approach for detecting source code plagiarism. *IAENG International Journal of Computer Science*, *44*(4), 501–522.

Karnalim, O. (2019). Source code plagiarism detection with low-level structural representation and information retrieval. *International Journal of Computers and Applications*. https://doi.org/10.1080/1206212X.2019.1589944

Karnalim, O., & Budi, S. (2018). The effectiveness of low-level structure-based approach toward source code plagiarism level taxonomy. *Sixth International Conference on Information and Communication Technology*, 130–134. https://doi.org/10.1109/ICoICT.2018.8528768

Karnalim, O., Budi, S., Toba, H., & Joy, M. (2019). Source code plagiarism dataset. Retrieved from https://github.com/oscarkarnalim/sourcecodeplagiarismdataset

Kermek, D., & Novak, M. (2016). Process model improvement for source code plagiarism detection in student programming assignments. *Informatics in Education*, *15*(1), 103–126. https://doi.org/10.15388/infedu.2016.06

Kuo, J.-Y., Cheng, H.-K., & Wang, P.-F. (2018). Program plagiarism detection with dynamic structure. *Seventh International Symposium on Next Generation Electronics*, 1–3. https://doi.org/10.1109/ISNE.2018.8394758

Kustanto, C., & Liem, I. (2009). Automatic source code plagiarism detection. *10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, 481–486. https://doi.org/10.1109/SNPD.2009.62

Levitin, A. (2012). *Introduction to the design & analysis of algorithms*. Pearson.

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

Liu, C., Chen, C., Han, J., & Yu, P. S. (2006). Gplag: detection of software plagiarism by program dependence graph analysis. *12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 872. https://doi.org/10.1145/1150402.1150522

Mirza, O. M., Joy, M., & Cosma, G. (2017). Style analysis for source code plagiarism detection – an analysis

of a dataset of student coursework. *17th International Conference on Advanced Learning Technologies*, 296–297. https://doi.org/10.1109/ICALT.2017.117

Moussiades, L., & Vakali, A. (2005). PDetect: a clustering approach for detecting plagiarism in source code datasets. *Computer Journal*, *48*(6), 651–661. https://doi.org/10.1093/comjnl/bxh119

Novak, M. (2016). Review of source-code plagiarism detection in academia. *39th International Convention on Information and Communication Technology, Electronics and Microelectronics*, 796–801. https://doi.org/10.1109/MIPRO.2016.7522248

Novak, M., Joy, M., & Kermek, D. (2019). Source-code similarity detection and detection tools Used in academia: a systematic review. *ACM Transactions on Computing Education*, *19*(3), 27:1--27:37. https://doi.org/10.1145/3313290

Ohmann, T., & Rahal, I. (2015). Efficient clustering-based source code plagiarism detection using PIY. *Knowledge and Information Systems*, *43*(2), 445–472. https://doi.org/10.1007/s10115-014-0742-2

Ottenstein, K. J. (1976). An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, *8*(4), 30–41. https://doi.org/10.1145/382222.382462

Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

Pawelczak, D. (2018). Benefits and drawbacks of source code plagiarism detection in engineering education. *2018 IEEE Global Engineering Education Conference (EDUCON)*, 1048–1056. https://doi.org/10.1109/EDUCON.2018.8363346

Poon, J. Y. H., Sugiyama, K., Tan, Y. F., & Kan, M.-Y. (2012). Instructor-centric source code plagiarism detection and plagiarism corpus. *17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, 122. https://doi.org/10.1145/2325296.2325328

Portillo-Dominguez, A. O., Ayala-Rivera, V., Murphy, E., & Murphy, J. (2017). A unified approach to automate the usage of plagiarism detection tools in programming courses. *12th International Conference on Computer Science and Education*, 18–23. https://doi.org/10.1109/ICCSE.2017.8085456

Prechelt, L., Malpohl, G., & Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, *8*(11), 1016–1038.

Rabbani, F. S., & Karnalim, O. (2017). Detecting source code plagiarism on .NET programming languages using low-level representation and adaptive local alignment. *Journal of Information and Organizational Sciences*, *41*(1), 105–123. https://doi.org/10.31341/jios.41.1.7

Simon, Sheard, J., Morgan, M., Petersen, A., Settle, A., & Sinclair, J. (2018). Informing students about academic integrity in programming. *20th Australasian Computing Education Conference*, 113–122. https://doi.org/10.1145/3160489.3160502

Smith, T. F., & Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, *147*(1), 195–197. https://doi.org/10.1016/0022-2836(81)90087-5

Sulistiani, L., & Karnalim, O. (2019). ES-Plag: efficient and sensitive source code plagiarism detection tool for academic environment. *Computer Applications in Engineering Education*, *27*(1), 166–182. https://doi.org/10.1002/cae.22066

Sun, W., Wang, X., Wu, H., Duan, D., Sun, Z., & Chen, Z. (2019). MAF: method-anchored test fragmentation for test code plagiarism detection. *41st International Conference on Software Engineering: Software Engineering Education and Training*, 110–120. https://doi.org/10.1109/ICSE-SEET.2019.00020

Ullah, F., Wang, J., Farhan, M., Jabbar, S., Wu, Z., & Khalid, S. (2018). Plagiarism detection in students' programming assignments based on semantics: multimedia e-learning based smart assessment methodology. *Multimedia Tools and Applications*. https://doi.org/10.1007/s11042-018-5827-6

Verco, K. L., & Wise, M. J. (1996). Software for detecting suspected plagiarism: comparing structure and attribute-counting systems. *First Australasian Conference on Computer Science Education*, 81–88. https://doi.org/10.1145/369585.369598

Wise, M. J. (1996). YAP3: improved detection of similarities in computer program and other texts. *27th SIGCSE Technical Symposium on Computer Science Education*, *28*(1), 130–134. https://doi.org/10.1145/236452.236525

Yang, F.-P., Jiau, H. C., & Ssu, K.-F. (2014). Beyond plagiarism: an active learning method to analyze causes behind code-similarity. *Computers & Education*, *70*, 161–172. https://doi.org/10.1016/J.COMPEDU.2013.08.005

Zhang, D., Joy, M., Cosma, G., Boyatt, R., Sinclair, J., & Yau, J. (2014). Source-code plagiarism in universities: a comparative study of student perspectives in China and the UK. *Assessment & Evaluation in Higher Education*, *39*(6), 743–758. https://doi.org/10.1080/02602938.2013.870122

Zhong, L., Wan, W., & Kong, D. (2016). Javaweb login authentication based on improved MD5 algorithm. *2016 International Conference on Audio, Language and Image Processing (ICALIP)*, 131–135. https://doi.org/10.1109/ICALIP.2016.7846653

O. 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 research interests are about computer science education, especially source code plagiarism and educational tools. He works at Maranatha Christian University as a full-time lecturer and he is currently pursuing a PhD in Information Technology at University of Newcastle, Australia.

S. Budi completed his academic exercise in Computer Science at the University of Tasmania, Australia. Australia Awards Scholarships and Sense-T Elite Scholarships enabled him to get his Master and PhD qualifications. His primary research interests include optimisation problem, environmental monitoring, data science, educational data mining, and computer vision.

H. Toba graduated with a Master of Science from Delft University Technology, the Netherlands in 2002, and completed his doctoral degree at Universitas Indonesia in 2015. He has been working as faculty members at the Faculty of Information Technology, Maranatha Christian University since 2003. His interests are in the field of information retrieval and educational datamining.

M. Joy received the MA degree in mathematics from Cambridge University, the MA degree in post-compulsory education from the University of Warwick, and the PhD degree in computer science from the University of East Anglia. He is currently a professor at the University of Warwick. His research interests focus on educational technology and computer science education.