Inf Retrieval J (2018) 21:1–23
https://doi.org/10.1007/s10791-017-9313-y

Retrieving and classifying instances of source code plagiarism

Debasis Ganguly[1] · Gareth J. F. Jones[1] · Aarón Ramírez-de-la-Cruz[2] · Gabriela Ramírez-de-la-Rosa[2] · Esaú Villatoro-Tello[2]

Received: 3 August 2016 / Accepted: 25 July 2017 / Published online: 13 September 2017

Abstract Automatic detection of source code plagiarism is an important research field for both the commercial software industry and within the research community. Existing methods of plagiarism detection primarily involve exhaustive pairwise document comparison, which does not scale well for large software collections. To achieve scalability, we approach the problem from an information retrieval (IR) perspective. We retrieve a ranked list of candidate documents in response to a pseudo-query representation constructed from each source code document in the collection. The challenge in source code document retrieval is that the standard bag-of-words (BoW) representation model for such documents is likely to result in many false positives being retrieved, because of the use of identical programming language specific constructs and keywords. To address this problem, we make use of an abstract syntax tree (AST) representation of the source code documents. While the IR approach is efficient, it is essentially unsupervised in nature. To further improve its effectiveness, we apply a supervised classifier (pre-trained with features extracted from sample plagiarized source code pairs) on the top ranked retrieved documents. We report experiments on the SOCO-2014 dataset comprising 12K Java source files with almost 1M lines of code. Our experiments confirm that the AST based approach

✉ Debasis Ganguly
debasis.ganguly1@ie.ibm.com

Gareth J. F. Jones
gjones@computing.dcu.ie

Aarón Ramírez-de-la-Cruz
aaron.rc24@gmail.com

Gabriela Ramírez-de-la-Rosa
gramirez@correo.cua.uam.mx

Esaú Villatoro-Tello
evillatoro@correo.cua.uam.mx

[1] ADAPT Centre, School of Computing, Dublin City University, Dublin, Ireland

[2] Language and Reasoning Research Group, Information Technologies Department, Universidad Autónoma Metropolitana, Cuajimalpa, México, D.F., Mexico

produces significantly better retrieval effectiveness than a standard BoW representation, i.e., the AST based approach is able to identify a higher number of plagiarized source code documents at top ranks in response to a query source code document. The supervised classifier, trained on features extracted from sample plagiarized source code pairs, is shown to effectively filter and thus further improve the ranked list of retrieved candidate plagiarized documents.

Keywords Source code plagiarism detection · Field based indexing and retrieval · Lexical, Structural and stylistic features · Document representation

1 Introduction

Plagiarism detection for programming language source code represents a topic of growing interest for both the software industry and academia. While in the case of the former, automated plagiarism detection helps prevent copyright infringements,[1] for the latter it finds use in minimizing unauthorized source code reuse in programming assignments among students. Additionally, community question answering (CQA) forums and programming related blogs on the web, e.g., StackOverflow and "Google Code", have made source code widely available to be read, copied and modified. Programmers often reuse source code snippets available on the web without acknowledging their true sources. The massive amount of source code reuse makes it impractical to perform manual analysis of unauthorized source codes. This necessitates the application of automated methods for source code plagiarism detection. Moreover, whilst not strictly plagiarism, automated source code similarity detection can also be beneficial to maximize the legal code reuse practices within a software company, for example an automatic tool can suggest to a programmer the use of a piece of previously developed code as a library instead of copy-pasting it. Consequently, automated source code plagiarism detection has become an important research topic.

Existing automated source code plagiarism detection systems mostly seek to detect plagiarism cases by exhaustive comparison. While this approach is feasible for small source code collections, such as those encountered in local checking for source code reuse in student assignments, the quadratic time complexity of the pairwise computations makes it unsuitable for larger collections, such as the Google code base.

In this work, we address the computational overhead of the exhaustive pairwise computation approach by using an information retrieval (IR) based method. In this approach, original source code documents (i.e., files from known authors) are added to an inverted list based indexed organization. Thus, every time a new source code file arrives (i.e., a suspicious file), this is treated as a pseudo-query to retrieve a ranked list of similar documents from the collection. Retrieved candidate original documents are then selected for further examination though a supervised classifier that accurately identifies plagiarized source code documents.

However, for source code documents, the computation of inter-document similarities using a traditional bag-of-words (BoW) encoding is likely to result in a massive number of false *hits* (non-zero similarity values) of non-plagiarized code due to the frequent use of

[1] Refer to the case of Oracle America, Inc. versus Google, Inc. for an example of possible source code plagiarism: https://www.theguardian.com/technology/2016/may/26/google-wins-copyright-lawsuit-oracle-java-code, http://www.potomaclaw.com/oracle-v-google-copyrightability-apis/.

identical constructs and keywords within code segments for a particular programming language.

Consequently, realizing an effective means of source code plagiarism detection based on an IR approach raises its own research challenges.

Q1: Can a BoW model (as used in standard IR) suffice for effective source code plagiarism detection, and can the source code structure be used to contribute to this effectiveness, for example to minimize the number of false positive retrieved items?
Q2: Can source code documents be indexed so that a retrieval model can best make use of the indexed terms to retrieve relevant (plagiarized) documents at top ranks?
Q3: How should a source code document be represented as a pseudo-query for effective use in an IR-based method?

An IR approach can potentially identify a set of candidate relevant (plagiarized) documents very quickly from a large collection of source code documents. However, one major limitation of the retrieval approach is that it is essentially unsupervised, and hence cannot provide the opportunity to benefit from manually annotated source code plagiarism sample pairs when they are available. It is intuitive that in such cases a supervised learning algorithm that utilizes fine-grained sub-document level characteristic features, when applied over the candidate documents obtained with the IR approach, could provide added benefit. Moreover, similar to text documents in natural language (e.g., news), source code documents in addition to containing structural and semantic information, also possess inherent characteristic signatures of the original author's writing style, such as indentation style, variable naming convention etc., which may serve as potentially useful features to train a supervised classifier for distinguishing true plagiarism cases from false ones.

In this study, we propose a supervised classification approach which makes use of features derived from source code documents such as character n-grams, data types, and identifier names. With the help of such a broad set of specific features, we aim to capture some of the most common practices employed by a plagiarist when attempting to camouflage plagiarized sections. Our method considers different characteristic features at multiple levels of granularity for the source code plagiarism detection task.

Generally speaking, the architectural configuration of our proposed two stage method can be described as follows. The first stage comprises of an efficient processing approach for fast detection of candidate plagiarized source codes with the help of an IR based approach. In the second stage, a supervised approach is employed for a more fine-grained analysis over the candidate plagiarized documents. We might think of this as akin to re-ranking in standard IR. In fact, such a two stage process for plagiarism detection has been found to be effective for natural language text documents as well (Stein et al. 2011; Potthast et al. 2014). However, in the context of our problem, a bag-of-words based document representation for an IR based approach is not able to capture the inherent grammatical structure of source codes. The main difference between our work and the two stage pipeline of plagiarism detection for natural language documents, as reported in (Potthast et al. 2014), is the incorporation of the abstract syntax tree (AST) information in document index construction and its use during the retrieval phase.

The remainder of this paper is organized as follows. In Sect. 2, we report previous research on source code plagiarism detection. In Sect. 3, we describe our proposed IR approach for identifying a candidate plagiarized documents. In Sect. 4, we introduce the features of our classification approach for predicting plagiarized documents from the candidate ones obtained with initial retrieval. Section 5 describes the experimental

environment and Sect. 6 reports the results of our experiments. Finally, Section 7 concludes the paper with suggestions for further elaboration of this work.

2 Related work

In this section we give a concise review of the most relevant existing work relating to identification of plagiarised source code. Generally speaking, two main directions have been proposed for tackling the source code plagiarism detection problem, namely natural language processing (NLP) based methods and IR based approaches.

The most common approach to software plagiarism detection involves application of NLP adapted to the specific characteristics of source code files. One such method takes into account the "whitespace" indentation patterns of a source code file (Baer and Zeidman 2012), where a source code document is converted to a *pattern*, namely whitespace format, replacing any visible character by `X` and any whitespace by `S`, and leaving newlines as they appear. The similarity between two source code documents is then computed by the longest common substring (LCS) between the two patterns.

Another common approach to source code plagiarism detection is to determine the fingerprint of a source code document by making use of the word n-grams. Thus, in (Marinescu et al. 2012) authors proposed representing different source code segments with hashes,[2] which are compared using the Winnowing algorithm (Schleimer et al. 2003). However, this approach does not consider important characteristics inherent to source code such as keywords, identifiers names, number of lines, number of terms per line, number of hapaxes, etc. For instance, the work reported in (Narayanan and Simi 2012) proposes a similarity measure that uses a particular weighting scheme for combining different characteristics of source codes. The work in (Cosma and Joy 2013) investigates dimensionality reduction with the help of latent semantic analysis (LSA). A major drawback of this work is that the experiments were conducted on small collections of documents (number of documents varying from 106 to 179), which is far from a real-life scenario. Contrastingly, we perform retrieval experiments on a much larger collection. A further limitation of the experimental setup in (Cosma and Joy 2013) is that the authors performed plagiarism detection experiments on four separate chunks of document collection pairs by using a priori knowledge that there are no plagiarized pairs between different collections. In contrast, we perform our experiments on a single collection without any a priori knowledge.

A major drawback of most previous research is that it uses methods that involve an exhaustive pair-wise similarity comparison between all source code files in a collection (Baxter et al. 1998; Chae et al. 2013a, b, thus making it too costly for real-life large sized collections. Due to the difficulty in carrying out experimental investigations on large software collections, such approaches have mostly been evaluated on very small collections of source code, e.g., the evaluation in (Chae et al. 2013a) uses a collection of 56 programs and the evaluation in (Baxter et al. 1998) is carried out on $40K$ lines of code. In contrast, the dataset used for our experiments is comprised of over $12K$ source code documents and about 1M lines of code (see Sect. 5 for more details). Further, it is worth mentioning that most of the previous research employs its own manually constructed corpus for evaluation purposes, meaning that it is not possible to obtain conclusive and fair comparisons between existing approaches and proposed new ones. Contrastingly, our

[2] http://theory.stanford.edu/~aiken/moss/.

experiments are conducted on a publicly available dataset (described in Sect. 5), thus ensuring reproducibility of the experiments.

Moreover, it should be noted that a common trend within existing work is that it does not differentiate between structural (pertaining to source code syntax) and stylistic features (pertaining to individual coding styles and patterns). However, it is intuitive that each individual feature, i.e., either structural or stylistic, provides its own important information that should perhaps not be mixed with the other. In our work, we handle these two classes of features differently.

Finally, taking an alternative approach relevant to our study, the work reported in (Burrows et al. 2007) develops an IR based approach to software code plagiarism, where the type of each token (such as name, data type, etc.) in a source code document is concatenated to form a string, character *n*-grams of which are then stored in an index. A key difference between our work and this earlier work is that the latter does not store the identifier (function or data variable) names in the index and does not make use of the structure of the source code document with the help of fields.

In summary, most related existing work has applied a pair-wise comparison between source code files, which is an unsuitable approach for a real life scenario. Accordingly, several similarity measures are employed in order to highlight the content overlap degree between a pair of source code files. In contrast to previous work, in this paper we tackle the problem of computational cost by means of an IR based approach which makes use of an abstract syntax tree (AST) representation for the index construction. Then, to further improve the effectiveness of the method we apply a supervised method, which represents a pair of source code files using three sets of high-level attributes, namely lexical, structural and stylistic features. The remaining sections of this paper describe our proposed approach and our experimental investigation of its effectiveness.

3 Retrieval stage

In this section, we first motivate our approach by outlining deficiencies of standard approaches for the source code plagiarism detection problem. We then describe our proposed document representation method for indexing and retrieval of source code.

3.1 Motivation

A standard BoW encoding of source code is likely to result in falsely high similarity values between non-plagiarized document pairs due to the use of similar programming language specific constructs and keywords. Figures 1 and 2 illustrate this point with an example. Figure 1a shows a very simple Java program[3] for adding two numbers. Consider the problem of subtracting two numbers, shown in Fig. 1b, c. The code in Fig. 1b essentially copies the content from that of 1a with slight modifications in the function names and the operator, and hence is a typical instance of plagiarism. Contrastingly, the code in Fig. 1c illustrates the ideal way of code reuse which is that of calling a public method from an open source, and hence should definitely not be considered as plagiarism.

Ideally speaking, the similarity between two source code documents for the purpose of plagiarism detection should take into account the extent of overlap in the structures of their

[3] The code fragments of Fig. 1 are motivating examples only, and do not form a part of our experimental dataset.

```
class Add {
    static int add(int a, int b) {
        return a+b;
    }
}
```

(a)

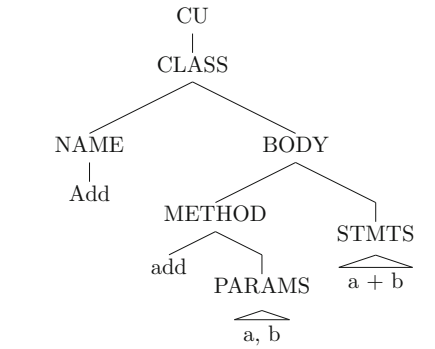
```
class Sub {
    static int sub(int a, int b) {
        return a-b;
    }
}
```

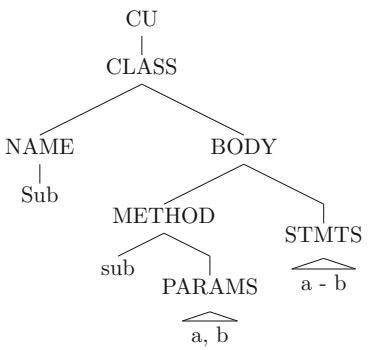
(b)

```
class Sub2 {
    static int sub(int a, int b) {
        int res = Add.add(a, -b);
        return res;
    }
}
```

(c)

Fig. 1 An example Java code fragment showing two sample ways of reusing it. While the programmer of **b** essentially copies the code from (**a**), the program of **c** shows a valid reuse case of invoking a method from (**a**).





(class, 1) (Add, 1) (static, 1) (int, 3) (add, 1) (a, 2), (b, 2), (return, 1) (plus, 1)

(class, 1) (Sub, 1) (static, 1) (int, 3) (sub, 1) (a, 2), (b, 2), (return, 1) (minus, 1)

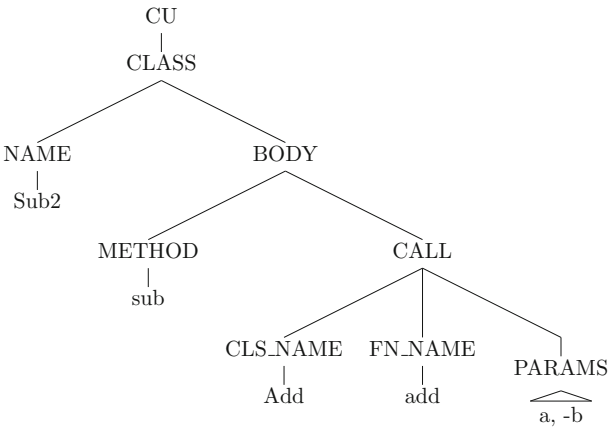
Field Name	Terms
Classes	Add
Method calls	
String literals	
Arrays	
Method definitions	add a b
Assignment statements	a plus b
Package imports	
Comments	

(a)

Field Name	Terms
Classes	Sub
Method calls	
String literals	
Arrays	
Method definitions	sub a b
Assignment statements	a minus b
Package imports	
Comments	

(b)

Fig. 2 Abstract syntax tree (AST) outline, BoW, and per-field BoW representation of the code fragments from Fig. 1a, b . The cosine similarity between the programs of Fig. 1a, b ($sim(a, b) = 0.869$) is computed using the BoW representation



(class, 1) (Sub2, 1) (static, 1) (int, 4) (sub, 1) (a, 2), (b, 2), (return, 1) (minus, 1) (**Add**, 1) (**add**, 1) (res, 1)

Field Name	Terms
Classes	Sub2
Method calls	Add add a minus b
String literals	
Arrays	
Method definitions	sub a b
Assignment statements	res a minus b
Package imports	
Comments	

Fig. 3 Cosine similarity between the programs of Fig. 1a, c ($sim(a, c) = 0.907$) shows how the program in Fig. 1c can be falsely identified as a plagiarism case with the BoW representation

abstract syntax trees (AST). In fact, it can be seen that the ASTs between Fig. 2a, b are more similar than those between Figs. 2a and 3. However, exhaustive pairwise computation of tree distances between source code parse trees is not computationally realistic, and hence a suitable approximating measure should be devised for this similarity comparison if it is to be scaled to real world tasks.

A simple approach to developing an approximate comparison between source codes would be to use a BoW model for the source code document representation. However, this may not be a suitable approximating function given the high overlap of common ground among source code files. To see the problem of the BoW model, note that the additional words "Add" and "add"[4] shown in bold face in Fig. 3, contribute to additional matching terms, which in turn increases the cosine similarity of this code with that of 2a. In addition, programs tend to use a frequent set of variable names, specially for looping constructs, e.g., *i*, *j*, *k*, etc., which may also cause false high similarities. Furthermore, programs make extensive use of common library classes, such as the "ArrayList", "HashMap", etc. in Java, which may also contribute to false matches. For example, two Java programs making

[4] According to Java programming conventions, an identifier name starting with an upper case letter denotes a 'class' name, whereas one beginning with a lower case denotes the name of a variable or a method, hence, "Add" and "add" have different semantic meanings.

use of the standard library class "HashMap" may be falsely identified as a plagiarized pair. The next subsection describes how we tackled this problem.

3.2 Proposed IR approach

A standard way to avoid per pair similarity computation is to use an inverted list organization of documents to retrieve a candidate list of the top most similar documents with respect to a given query, as classically applied in IR systems. In this section, we describe how such an approach can be applied for the source code plagiarism detection task.

3.2.1 Document representation

Given a suspicious source code file or even a small fragment of source code suspected of being plagiarised, we need to identify all the candidate (original) source code files within a collection of source codes of known authorship from which it may have been copied. Thus, we propose to treat the suspicious document as a pseudo-query and retrieve a list of the top ranked most similar documents in response to the query. However, in contrast to the standard method of result presentation as a ranked list in IR, the objective in the case of source code plagiarism detection is to obtain a set of candidate files that are likely to be plagiarized.

As discussed in Sect. 3.1, the BoW document model representation of a source code file cannot effectively capture cases where a part of the source code of one program is copy-pasted into another one. To alleviate this problem, a solution is to take into account the grammatical structure of a source code while representing the source document as a vector. The standard way of representing a program control flow structure is through an AST (abstract syntax tree) (Jones 2003). ASTs are typically used for static code analysis, i.e., detecting whether parts of programs produce identical outputs when presented with identical inputs (Baxter et al. 1998; Neamtiu et al. 2005). In our case, the AST can be applied for the plagiarism detection problem because the ASTs of parts of cut-pasted source codes have similar structures with respect to the original code, (e.g., see Fig. 2).

Unfortunately, there is no direct way of computing AST similarity (e.g., the edit distance between two ASTs) within the inverted list organization used as the standard data structure in IR. A solution is to decompose a whole source code document into smaller fragments (commonly known as *fields* in the IR literature). Most natural language text documents are explicitly structured into different fields, e.g., title, abstract, body, etc. It is a common practice in IR to compute document-query similarities on a per-field basis and then to combine the per-field scores to get the overall query-document similarity (Ogilvie and Callan 2003; Kim and Croft 2012). Field based retrieval models have two helpful properties in the context of source code comparison. Firstly, they allow consideration of only intra field matches, while avoiding the inter field matching. Secondly, they allow for provision of per-field normalization.

In our case, the source code documents are not explicitly marked into separate sections or fields. The field representation of a source code document can however be obtained from its AST. After constructing an AST from each source code document in the collection, we visit and extract terms from a specific subset of terminal (leaf level) nodes of the AST. These terms are then stored in separate fields corresponding to the nodes from which they were extracted. In our experiments, to simplify the experimental setup, the field weights are set uniformly across all the fields.

Table 1 AST nodes of a Java program from which terms are extracted during indexing

Field name	Field description
Classes	Names of Java classes
Method calls	Method names with actual parameter names and types
String literals	Values of the string constants
Arrays	Names of arrays and dimensions
Method definitions	Names of methods and formal parameter names and types
Assignment statements	Variable names and types
Package imports	Names of imported packages
Comments	Text inside comments

The specific nodes of the AST that we make use of are shown in Table 1 (see also Figs. 2 and 3 which show the field names at the bottom of the corresponding ASTs). For example, with reference to Fig. 3, the field "Classes" is comprised of the term "Sub2" because there is only one class declaration in this source file, which is named "Sub2" (see Fig. 1c). The list of fields that we use for plagiarism detection typically represents the major Java program constructs that can be used as cues for detecting plagiarized cases.

A field representation of a document is expected to better utilize the source code document structure than a flat BoW representation. This is shown in Fig. 1b, c, where it is seen that the additional matches (i.e., "Add" and "add" shown bold faced in Fig. 1c) do not contribute to the overall similarity because these words occur in different fields ("Classes" and "Method calls" in Fig. 1a, c respectively). To give another example, a match in the "String literals" field will be treated separately with respect to a match in the "Classes" field, as a result of which a program using the string constant "HelloWorld" will not be considered to have been plagiarized from a source which defines a class named "HelloWorld".

3.2.2 *Query representation*

Since only a part of the source code is typically copy-pasted into another program, it is not generally appropriate to use whole source code documents as a pseudo-queries. Instead, we propose to extract a preset number of terms from each field of a document (see Table 1) to construct a pseudo-query. This approach of extracting a selected number of terms from each field of a document has been applied to formulate queries from expository articles such as patents (Takaki et al. 2004; Xue and Croft 2009). In the case of source code plagiarism detection, this way of formulating queries ensures that representative terms from each individual field, e.g., classes, method calls, etc. (see Table 1), of the current suspicious source code (the pseudo-query) are matched with individual fields of indexed documents in the collection. We use the field language model (LM), shown in Eq. 1, as the term selection function (Hiemstra 2000; Ponte 1998) to obtain representative terms from each field.

$$LM(t,f,d) = \lambda \frac{tf(t,f,d)}{len(f,d)} + (1-\lambda)\frac{cf(t)}{cs} \quad (1)$$

In order to formulate a query from a document d, we score each term of each field f of d by the function shown in Eq. 1, and then select the top most k terms, where k is a parameter to

represent the query constructed from d. The value of the parameter k, used for the query representation for each document, is determined with the help of a grid search in our experiments. Although small constant values of k, irrespective of the document length, may not be sufficient to represent large source code documents, such a query representation mechanism with a fixed number of terms ensures consistent retrieval efficiency for all documents in the collection.

The parameter λ controls the relative importance of the term frequency of a term t in field f of document d, $tf(t, f, d)$, as against the collection frequency of the term, $cf(t)$, normalized by dividing it by the collection size, cs. The expression $len(f, d)$ denotes the length of the field f in document d.

We use relative term counts instead of absolute ones, because completely ignoring the document length factor may produce non-representative terms in the query, such as a local variable name with a very small scope. The motivation for using collection statistics of a term, in addition to its term frequency, is explained by the observation that some programming language specific words, such as keywords (e.g., *for*, *while*) or common variable names (e.g., i, j etc.), are used very frequently and hence a match for these terms may not strongly indicate potential plagiarism. On the other hand, a match for rare terms, e.g., uncommon variable or function names, should represent more likely cases of plagiarism.

4 Classification stage

In this section, we describe our proposed method for performing the more fine-grained analysis of the retrieved source code documents. The IR stage (discussed in Sect. 3) returns a ranked list of candidate (plagiarized) source code files ordered by their similarity to the suspicious source code (i.e., the query document). However, as discussed earlier, a high retrieval similarity score does not necessarily mean a high likelihood of plagiarism. To address this issue, we use a supervised classification approach that outputs a binary decision of whether a document in the candidate list of retrieved documents is plagiarized or not. In particular, in order to capture some of the most common practices among source code plagiarists, our classifier makes use of three sets of high-level features namely: lexical, structural and stylistic, each of which is explained in the subsequent subsections.

4.1 Lexical features

The main idea of the lexical features (Flores et al. 2014a)is to represent a source code fragment by means of a bag of character n-grams, with the intention of measuring lexical similarity (i.e., content overlap). In particular, we use the value of $n = 3$ for character grams, because previous research has shown that character 3-g cosine similarity proves to be the most effective for detecting plagiarized pairs (Flores et al. 2011). The main problem with the method described in (Flores et al. 2011) is related to the number of keywords present in any source code document, resulting in an overestimation of the lexical similarity. Therefore, and contrary to previous work to alleviate this problem, we eliminate all reserved words from the source code documents, hence the lexical similarity will not result in bias due to overlap between keywords.

4.2 Structural features

The proposed structural features (six in total) consist of two forms of representation, both of which are based on a function's signature definition within a source code. The first types of representation consider the data types (e.g.,, `int, char, long, float, String`, etc.) of a function's signatures.[5] The second type of representation considers the identifiers, e.g., a variable's name used by the programmer in the function's signature, i.e., the name of the function itself and the names of all its arguments. Intuitively, the former attempts to capture those cases where the programmer changes the identifiers of some function in order to camouflage the plagiarism, but not the data types; whereas the latter is able to capture those cases where the plagiarist changes a function's data types but not the names of the identifiers. As stated in (Faidhi and Robinson 1987) these type of practices represent different forms of plagiarism.

4.2.1 Similarity between data types

We represent each function's signature as a list of data types. For example, the following function's signature `int sum(int numX, int numY)` is transformed into `int (int, int)`. Our proposed representation also accounts for the frequency of each data type. To calculate the similarity between two functions, we need to compare the two elements from the signature of these functions, i.e., the return data type and the data types of the arguments. We measure the importance of each element independently to establish the best way to combine them in a second step.

Given two functions, m^s and m^c, contained in the suspicious code (d_s) and the candidate code (d_c) respectively, the similarity between their return data types, sim_r, is 1 if the return types are identical or 0 otherwise. Next, to determine the similarity of their arguments' data types, we propose a more elaborate strategy. First, we compose a bag of data types for each function with their respective frequencies; hence each function is represented as a vector where its components are data types. Then, we compute a similarity between two functions' vectors $\mathbf{m^s}$ and $\mathbf{m^c}$ as defined in Eq. 2, where n indicates the number of different data types in both functions, i.e., the vocabulary of data types. Notice that Eq. 2 depicts the Jaccard coefficient, and accounts for the data type coincidences between two functions.

$$sim_a(\mathbf{m^s}, \mathbf{m^c}) = \frac{\sum_{i=0}^{n} \min(\mathbf{m^s}_i, \mathbf{m^c}_i)}{\sum_{i=0}^{n} \max(\mathbf{m^s}_i, \mathbf{m^c}_i)} \tag{2}$$

After the return data type similarity (sim_r) and the arguments' data type similarity (sim_a) are computed, we determine a single value (sim_1) of the data type similarity by means of a linear combination as defined in Eq. 3, where $\sigma \in \mathbb{R}$ and $\sigma \in (0, 1)$. For our experimental investigation we set $\sigma = 0.5$, thus in effect, giving equal importance to return types and arguments. This value was empirically determined using the available training set (see Sect. 5.2).

$$sim_1(m^s, m^c) = \sigma * sim_r(m^s, m^c) + (1 - \sigma) * sim_a(\mathbf{m^s}, \mathbf{m^c}) \tag{3}$$

Finally, in order to compute the global *data-types* similarity between all functions defined within the source code pair d_s and d_c, we compose a function-similarity matrix $\mathbf{M}^{type}_{s,c}$,

[5] We refer to these simply as "functions" and do not make a distinction between a function and a class method.

where all functions contained in d_s are compared against those in d_c. Thus, the final value of similarity between a code pair is computed as shown in Eq. 4, where $f(x)$ represents either the maximum, minimum or the average of the matrix values.

$$sim_{DataTypes}(d_s, d_c) = f(\mathbf{M}_{s,c}^{type}) \tag{4}$$

Notice that selecting either the maximum or the minimum value from $\mathbf{M}_{s,c}^{type}$ implies that the similarity between d_s and d_c is determined by considering only the similarity of one pair of functions (the most similar or the least similar respectively), whilst the average value considers the similarity between all the possible pairs of functions contained in d_s and d_c. Since all of these (max/min/avg) may act as distinguishing features, we use them all in our feature set.

4.2.2 Similarity between identifiers

Complementary to the *data types* features, this set of features considers the structure by means of using a function's name as well as the name of its arguments. For this, we construct a string formed by the concatenation of a function's name and the name of all its arguments. All characters are converted to lowercase and white spaces are removed. For example, the function `int sum(int numX, int numY)` is represented as the string `sumnumxnumy`. The next step consists of computing the corresponding character *n*-grams representation using a binary weighting scheme, particularly, we set the value of $n = 3$. We based our decision on the results obtained in (Flores et al. 2011, 2014a), where exhaustive experimentation was done in order to resolve what is the best size of n in a character *n*-grams representation.

Accordingly, given the functions m^s and m^c, belonging to the suspicious (d_s) and to the candidate (d_c) source code respectively, and their corresponding sets of character 3-g, $\mathbf{m^s}$ and $\mathbf{m^c}$, we compute their similarity using the Jaccard coefficient as shown in Eq. 5. We employ the Jaccard coefficient since it is particularly well suited to handling asymmetric binary attributes, i.e., comparing the similarity and diversity of two sets.

$$sim_2(m^s, m^c) = \frac{\mathbf{m^s} \cap \mathbf{m^c}}{\mathbf{m^s} \cup \mathbf{m^c}} \tag{5}$$

Analogous to the *data type* features, every function in the suspicious source code d_s and in the candidate source code d_c are compared. From this, we obtain a *name-similarity* matrix $\mathbf{M}_{s,c}^{names}$. The overall similarity value of d_s and d_c is shown in Eq. 6.

$$sim_{Names}(d_s, d_c) = f(\mathbf{M}_{s,c}^{names}) \tag{6}$$

where $f(x)$ can denote either the maximum, minimum or the average of the matrix values. We include all these values in our feature set.

4.3 Stylistic features

Analogous to natural language text documents, which inherently contain author specific writing style characteristics (Grieve 2007), we hypothesize that source code also carries (to some extent) programmer specific stylistic features. Accordingly, we defined a set of 11 stylistic features appropriate for the type of documents under consideration, i.e., source code files. Based on inspection of code examples, we employ the number of lines of code,

the number of white spaces, the number of tabulations, the number of empty lines, the number of defined functions, average word length, the number of upper case letters, the number of lower case letters, the number of under scores, vocabulary size, and the lexical richness. The latter two features are defined as follows: on the one hand, the lexical richness is the ratio between the number of distinct lexical units and the total number of lexical elements used. This metric is intended to capture the vocabulary diversity used by the programmers. On the other hand, the vocabulary size represents the number of distinct lexical units employed by the programmer.

Our intuition is that all of these features might be helpful in distinguishing different programming styles. Finally, the stylistic similarity value between d_s and d_c is obtained by computing the cosine similarity between the pair of vectors representing the stylistic features of such documents.

5 Experimental setup

In this section, we describe the dataset and the tools used for our experiments.

5.1 Dataset

For the experiments reported in this paper, we used the test dataset of the Source Code reuse SOCO task, which was carried out as a part of the Forum of Information Retrieval Evaluation (FIRE 2014)[6] (Flores et al. 2014b). The documents in this dataset are real-life source code plagiarism cases. The collection[7] consists of annotated examples of source code files collected from the 2012 edition of the Google Code Jam Contest.[8]

An overview of the characteristics of the dataset is presented in Table 2. It can be seen that the entire collection of about 12, 000 files is categorized into three separate folder types, the first letter of the name indicating its type, i.e., “A”, “B” and “C”. The types themselves represent broad level categories of problems given to the Google CodeJam participants. Due to the large size of the corpus, it is practically impossible to obtain a per-pair manual plagiarism judgement in order to obtain a reference set for evaluation. Thus, to evaluate the performance of the participating systems during the SOCO track, the organizers applied a standard *pooling* approach (Sanderson and Zobel 2005). The intersection of the set of plagiarised pairs reported by the participating systems was used to construct the pool. The documents of this pool were then judged manually by the track organizers for manual relevance judgements, relevance of a document in this context referring to the scenario that it is being plagiarized from the query document source code.

It is important to mention that according to the SOCO task guidelines there are no plagiarism cases across different source code category types, e.g., it is known a priori that the sources belonging to category B1 are only plagiarized from those belonging to B1 and not from A1 or B2. However, this a priori knowledge fails to simulate a real-life collection accurately enough, because in a real-life situation, availability of such knowledge would be highly unlikely. Moreover, it also makes the task less challenging because there would be a lower number of documents to process in total. In our experiments, we ignore this a priori

[6] http://www.isical.ac.in/∼fire/.

[7] http://users.dsic.upv.es/grupos/nle/soco/.

[8] https://code.google.com/codejam/contest/1460488/dashboard

Table 2 Characteristics of the Java source code collection (categorized by theme)

Category	#Files	#Plagiarized	#Lines
Training set (classifier only)			
N/A	256	84	38,356
Test set (IR and classifier)			
A1	3241	54	266,885
A2	3093	47	249,263
B1	3268	73	247,696
B2	2266	34	174,416
C1	124	0	18,786
C2	88	14	12,194
Total	12,080	222	969,240

information in order to simulate a more realistic scenario. In our work, we process the whole collection of about 12,000 documents (without assuming that they are categorized into different thematic types). This adjustment to the task also helps to demonstrate the scalability of our IR based approach.

5.2 Training set for classification

The supervised approach, described in Sect. 4, requires a training set of manually annotated plagiarized source code pairs. To train our classifier, we used the official SOCO training dataset, which consists of 254 source code files where 84 are labeled as plagiarized pairs (See Table 2). By following this approach we ensure that the generated model is not being evaluated under the same set of documents, in other words, there is no overlap between training and test sets.

5.3 Settings

The AST for each Java source code was obtained with the help of "Java parser",[9] which is an open source syntax parser for Java programs. Information extracted from the AST nodes was then used to construct the field representation for every document in the index. We indexed the document set using Lucene (version 4.6),[10] The indexing and retrieval code is publicly available in the Github code repository.[11] To account for the non-informativeness of Java programming language specific words, we use a set of 25 frequent Java constructs as our stopword list.[12]

For supervised classification on the candidate documents obtained after retrieval, we employed the widely used machine learning toolkit Weka.[13] The classifier used was the state-of-the-art Random Forest (RF) classifier, which has been shown to perform better than other well known ones such as logistic regression and SVM (Fernández-Delgado et al.

[9] http://code.google.com/p/javaparser/.

[10] https://lucene.apache.org/core/4_6_0/index.html.

[11] https://github.com/gdebasis/YASOCS.

[12] https://github.com/gdebasis/YASOCS/blob/master/javastopwords.txt.

[13] http://www.cs.waikato.ac.nz/ml/weka/.

2014). The number of trees, which is a parameter of the RF classifier, was set to 10, as prescribed in (Breiman 2001), for all our classification experiments.

5.4 Evaluation metrics

To measure the effectiveness of the IR stage we use standard ranked list evaluation measures, such as the mean average precision (MAP), GMAP and recall. We emphasize that these metrics are only used to evaluate the IR stage of the plagiarism detection process. Clearly, if a higher number of 'relevant' (plagiarized) documents is retrieved within the top K ranks, it gives the subsequent filtering stage the potential to predict them as sources from which the current document is plagiarized.

While it is true that some source code documents have many plagiarized cases, whereas some do not, we emphasize that MAP (along with GMAP) reflects this scenario. This is because MAP combines aspects of both precision and recall, i.e., a high value of MAP represents the situation that firstly, a high number of 'relevant' documents are retrieved in the top K and that these are retrieved towards the top of the ranked list. Ideally speaking, a high MAP value in the IR stage would also indicate a better input to the supervised classification stage, because it would mean that the computationally expensive supervised stage can only work with a small subset of documents, thus contributing to increasing the efficiency of the overall process.

For measuring the overall effectiveness of the plagiarism detection process, we use standard set-based evaluation measures, such as precision, recall and F-score.

6 Results

This section reports the plagiarism detection results obtained with our proposed approach. We first evaluate the effectiveness of the IR stage (Sect. 6.1), which is followed by the evaluation of the classification stage (Sect. 6.2). Each IR run constitutes a ranked list of 100 documents, on which standard IR metrics are computed. The number of documents to be retrieved was empirically determined in previous experiments, specifically we found that a ranked list of 100 achieves nearly a perfect recall (see Table 3) and there is no requirement to go further down the ranked list. In addition, we compare the results of our experiments against the official runs submitted for the SOCO task (Sect. 6.3). Finally, we report the computational efficiency of our proposed method in the form of run-time comparisons (Sect. 6.4).

Table 3 Candidate plagiarized document detection with our proposed IR approach

Name	Parameters		Evaluation metrics		
	AST	Fld	MAP	GMAP	Recall
JPlag	n/a	n/a	0.2940	0.0004	0.2978
LM	No	No	0.5199	0.3160	0.9595
LM_AST	Yes	No	0.5274	0.3381	0.9595
LM_AST_SPLIT	Yes	No	0.3142	0.0014	0.4258
FLM_AST	Yes	Yes	**0.5345**	**0.4008**	**0.9865**

6.1 Run descriptions

This section describes the configuration employed for performing our experiments as well as how the baselines were defined.

JPlag (External Baseline) As an external baseline, we used the popular source code plagiarism tool, JPlag (Prechelt et al. 2002), using its default parameters (as was used in the SOCO task baseline). A reason for using this external baseline is to be able to compare our results with the SOCO submissions since this approach served as the baseline in the SOCO task as well (Flores et al. 2014c). In JPlag, after a source code is parsed and converted into token strings, a greedy string tiling algorithm is applied to identify the longest non-overlapped common sub-strings within the tokens, mostly similar to (Burrows et al. 2007).

Our Baselines As our own baselines, we conducted experiments with three different approaches. The first of these simply uses a standard language modeling (LM) retrieval model (Hiemstra 2000) with a flat BoW representation, named "LM" in Table 3. We used Jelinek-Mercer smoothing with λ set to 0.4, after tuning this parameter on the SOCO training collection of documents. In this baseline, the structural information of the source code documents is ignored. Source code documents are treated similar to non-structured text documents with the content in the index separated into separate fields.

As the second baseline approach, we extract the terms from the nodes (*fields*) of the AST, e.g., classes, method calls, etc. (see Table 1). However, we do not store these terms in separate fields; but again store them in a single field. This approach is denoted as "LM_AST". This method thus takes into consideration the structure of a source code for extracting key terms for indexing; but keeps storing a document in the index as a single BoW.

As a third baseline, we make use of the parse tree of a query source code document to decompose it into separate method bodies, each of which is treated as an individual query. The retrieved ranked lists for each of these is then merged using the standard COMBSUM fusion method (Fox et al. 1992). The rationale for using this approach is to account for the fact that sometimes a part of a source code is copied and pasted into another. The idea for this baseline is to use a combined aggregate of these piece-wise similarities to estimate the overall similarity between source code documents. We name this run "LM_AST_SPLIT".

AST based IR The fourth approach is our proposed method where we leverage the structural information of the source code documents by both extracting terms from the AST and storing them in different fields of the index, as described in Sect. 3.2. This is shown as "FLM_AST" in Table 3.

6.1.1 Results

The results obtained with different parameter settings are shown in Table 3. A number of findings can be observed from these results. First, it can be seen for our alternative systems that all the IR approaches, including the baselines, outperform the JPlag tool. This can be explained from the fact that JPlag ignores identifier names while converting a source code program into the BoW representation (Prechelt et al. 2002) (similar to Burrows et al. 2007). By contrast, all the IR approaches do consider the identifier names, which shows that identifier names play an important role in identifying candidate plagiarized documents. This is particularly true for relatively rare identifier names in the collection, because the IR approaches utilize the *idf* values of terms which the JPlag tool does not.

Second, it can be seen that splitting a document into individual queries (the method LM_AST_SPLIT) does not perform well in comparison to the approaches which construct a single pseudo-query form a source code document. A likely explanation for this observation is that the individual functions do not have sufficient information in them for identification of candidate plagiarism cases.

Third, the BoW baseline (LM) works well yielding a MAP of 0.5199, thus producing a strong baseline. However, the use of AST in term extraction for indexing (LM_AST) produces better results (increasing MAP to 0.5274) than LM (which ignores document structure). This suggests that the structure of a program can be a useful cue for this task.

Finally, it can be observed that the best results are obtained when terms extracted from the AST of a source code document are stored in different fields in the index and retrieved with the help of a field-based retrieval model (FLM_AST). The improvements in MAP obtained with our proposed method were found to be statistically significant (Wilcoxon test with 95% confidence measure) over the three baselines: 'LM', 'LM_AST' and 'LM_AST_SPLIT' (see Table 3). The improvements in GMAP are statistically significant as well for both the baselines, which is indicative that the per-query improvements are consistent.

In Fig. 4, we show the MAP values measured individually for each source code document category type with an intention to observe the per-category breakdown of retrieval effectiveness. It can be seen that the 'FLM_AST' method performs the best for each category (except 'A2').

The results shown in Table 3 were obtained by using whole documents as pseudo-queries for retrieval (i.e., all terms contained within documents). Instead of using whole documents as queries, it is more efficient (and often more effective) to use a selected number of terms as a pseudo-query for retrieving the documents. This way of using reduced queries for retrieval is particularly common for long documents, such as patents (Xue and Croft 2009), where a number of terms are extracted from each field of a document in order to construct the pseudo-query. The term scoring function to select the important or key terms from each field, as used in (Xue and Croft 2009), is the *tf-idf* score of a term. Since the retrieval model that we use for our experiments is LM, we use the LM

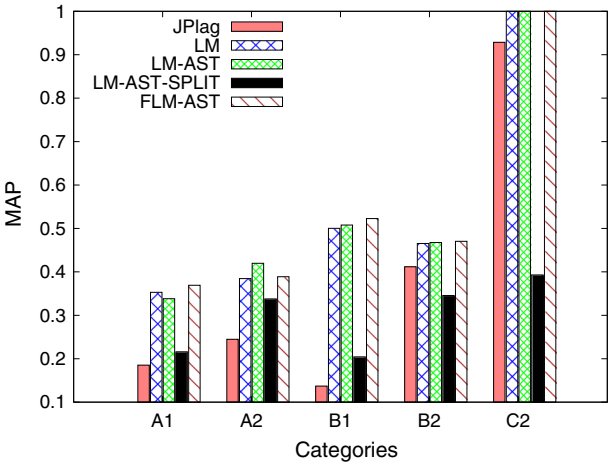


Fig. 4 Per category MAPs obtained with three different IR methods and the JPlag (c.f. Table 3)

term score, as shown in Eq. 1, to extract the key terms from each field of a source code document. The LM term score can in fact be considered as a generalized version of tf-idf, with the λ parameter being able to adjust the relative contributions of each component. The value of λ used for term selection was 0.4. Instead of tuning this parameter separately, we use the value of λ that was identical to the value used for the retrieval of documents.

In Table 4, we show the results obtained with the selective term extraction approach. The maximum number of terms to extract from each field of an indexed source code document is a parameter of this method. It can be observed that using at most 5 terms from each field produces the best results in terms of GMAP and recall. The effectiveness gradually decreases as the number of terms in the pseudo-query is increased. In terms of MAP, the best results are obtained with 20 terms.

6.2 Classification evaluation

As shown in the previous section, the most effective retrieval result in terms of MAP is obtained from 'FLM_AST' using 20 terms for the pseudo-query construction. We henceforth refer to this configuration as 'FLM_AST_20'. Consequently, for performing the experiments of the classification stage we use the retrieval results as obtained by the 'FLM_AST_20' configuration. The main goal of our experiments was to examine the ability of a supervised approach to further improve the effectiveness of plagiarized source code identification (see Sect. 4).

The classifier acts as a filter on the top k candidate documents obtained with FLM_AST_20, and outputs a filtered ranked list of documents in the following way. If a document is classified as plagiarized (the positive class) by the classifier, it is retained in the ranked list. On being classified as non-plagiarized (the negative class) the document is removed from the list. Note that the classifier only acts as a filter on the ranked list and does not re-rank its constituents. The objective is to remove false positive cases returned by the IR method.

Figure 5 shows the MAP values obtained after filtering the IR ranked list (FLM_AST_20). It can be seen that with the help of a supervised approach, the effectiveness of the results increases significantly. Particularly interesting is to see that the best results are obtained when only the top 20 documents from the retrieved ranked list are processed by the classifier. This shows that the documents classified as plagiarized are mostly reported within the top 20 rank by the retrieval method. The MAP values do not change beyond rank 20, which shows that the classifier does not identify any documents beyond rank 20 as plagiarized. Our classification experiments thus validate that it is not

Table 4 Results obtained when term extraction is applied for pseudo-query construction from source code documents

#Terms	MAP	GMAP	Recall
All	0.5345	0.4008	0.9865
5	0.5369	**0.4422**	**0.9955**
10	0.5367	0.4229	0.9909
20	**0.5371**	0.4232	0.9909
30	0.5348	0.4010	0.9864

First row (*All*) depicts results obtained with the FLM_AST configuration (see Table 3)

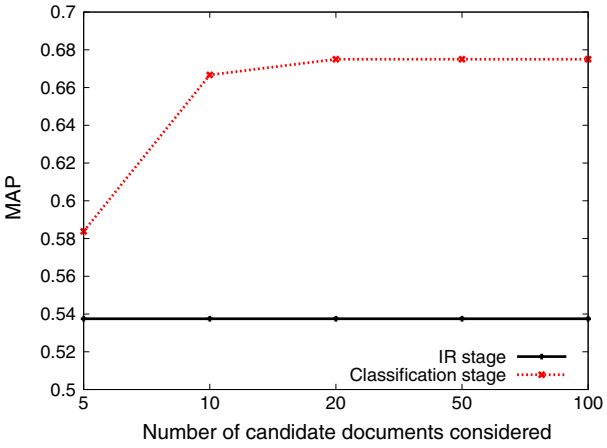


Fig. 5 MAP values obtained by classifying the candidate documents returned by the FLM_AST_20 retrieval model

necessary to apply a computationally intensive supervised classification step beyond a small number of very 'similar' candidate source code documents.

Finally, Table 5 reports the relative importance of features used in our classifier. For this we employed the well known *Information Gain* metric (*i-gain*) (Breiman 2001), which is an entropy based measure that helps in determining the importance from a set of features. It can be seen that the lexical feature (character 3-g cosine similarity) turns out to be the most distinguishing feature. The next set of important features are those involving the identifier names (Sect. 4.2). As expected, the features corresponding to the data types are less important than those involving identifier names, because a match in the function name denotes a more likely case of source code plagiarism than a match in the function prototype. The stylistic features (Sect. 4.3), which aim to capture the inherent writing styles of authors e.g., white space indentation, variable name length etc., are also important for source code plagiarism detection.

6.3 Comparison with SOCO official runs

In this section, we compare the results of our experiments with the official runs submitted for the SOCO task, the details of which can be found in the overview paper of the task (Flores et al. 2014c). In order to compare the results, we report the set based evaluation

Table 5 Feature ranking with *i-gain* measure

Feature name	*i-gain*
Lexical (character 3-g)	**0.191**
Avg identifier sim (Eq. 6)	0.051
Min identifier sim (Eq. 6)	0.039
Avg identifier sim (Eq. 6)	0.037
Stylistic	0.036
Min data type sim (Eq. 4)	0.034
Max data type sim (Eq. 4)	0.030
Avg data type sim (Eq. 4)	0.022

measures for our experiments as well. We emphasize that rank based measures reported in Sects. 6.1 and 6.2 still continue to hold as important metrics for this task.

In Table 6, we report the two baselines used by the SOCO organizers and the best of the submitted results in the SOCO task. The first baseline uses JPlag and hence is similar to our baseline reported in this paper. The second baseline uses a character 3-gram model weighted using term frequency and a cosine measure to compute the similarity between two source code document pairs. This baseline considers all source code pairs that exceed a similarity threshold of 0.95 to be plagiarized pairs. Finally, the last row depicts the best performance obtained during the SOCO competition.

In Table 7, we report the set based evaluation measures obtained in our experiments. It can be seen that the classification stage always improves the evaluation measures over the IR stage alone. For the IR stage, the ranked list of retrieved documents is cut-off at a fixed rank (20 in this case) to obtain the set representation. The IR stage achieves a high recall due to the fact that it is able to find (almost) all plagiarized documents within the top 20 positions in the ranked list (particularly the FLM_AST configuration). However, the precision is not satisfactory due to the presence of false positives. The results improve with the classifier because of the elimination of these false positives, which in turn also leads to improving the F-score significantly.

Out of the three retrieval approaches (followed by the same classification approach), FLM_AST performs the best because of the better retrieval quality (see Table 3). Notice that recall drops after applying the supervised classification step, because the supervised step aims to achieve a high precision (and in this trade-off process, recall decreases). A high precision is important from the end-user perspective because it indicates a reduced number of false positives being reported.

6.4 Efficiency evaluation

In this section, we report the computational run times of each individual method. The objective is to see how much we gain in terms of run-time, by applying our retrieval strategy. When reporting the run times, we do not take into consideration the content pre-processing times, i.e., the indexing time for the IR stage or the training time for the classification stage.

Figure 6 shows (using a logarithmic scale) the execution times required by the different approaches and their combinations. All the experiments were executed on a standard PC with a 3.16 GHz Quad-core CPU with 8GB RAM. To obtain run-time comparisons, we executed the RF classifier (see Sect. 6.2) on all pairs of documents. As expected, due to the quadratic time complexity this method takes a massive amount of time to execute, specifically about 3 days.

Notice that the constant valued horizontal line represents the total retrieval time taken by FLM_AST_20, i.e., the time to execute about 12, 000 queries on the collection of about

Table 6 Source Code Reuse (SOCO) official results (Flores et al. 2014b)

Run name	Set-based metrics		
	Precision	Recall	F-score
Baseline 1	0.542	0.293	0.38
Baseline 2	0.457	0.712	0.556
Best (UAM-C-run3)	**0.691**	**0.968**	**0.807**

Table 7 Experimental results evaluated with set based metrics (for comparison with SOCO runs)

Run name	IR (top 20 set)			IR + Classification		
	Prec	Rcll	F-sc	Prec	Rcll	F-sc
LM	0.432	0.995	0.602	0.808	0.964	0.879
LM_AST	0.530	0.995	0.692	0.856	0.964	0.907
FLM_AST	0.515	**1.000**	0.689	**0.874**	0.968	**0.919**

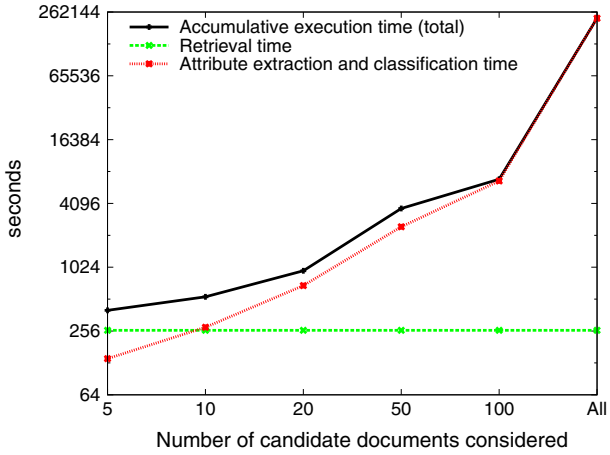


Fig. 6 Comparison of the execution times (on a log scale) for the different methods

12, 000 documents (see Table 2). The x-axis shows the number of top k documents used for classification. The IR retrieval time, included for the sake of comparison in Fig. 6, is shown as a constant because the IR method does not involve supervised classification.

It can be observed from Fig. 6 that the execution time involved in feature extraction and classification for the top k candidate source code documents steadily increases as more documents are processed for classification. The solid black line represents the accumulated execution time for the combined approach, i.e., IR + classification. It is important to mention that the combination method (which achieves the best results for $k = 20$ top documents) takes a significantly lower run-time than the traditional pair-wise similarity computation approach, which is shown by the top right point in the graph.

7 Conclusions and future work

In this paper we have described an approach to source code plagiarism detection which combines IR and supervised classification. The aim of the IR stage is to retrieve a set of potential plagiarism cases in an efficient manner. In the second stage, a supervised approach is employed to perform a more fine-grained analysis over the candidate set of plagiarized documents returned by the retrieval stage.

We hypothesized that the BoW model of indexed document representation may not be suitable for source codes (see research question Q1 in Sect. 1). Experiments confirm that approaches that utilize the source code structure are more effective in retrieving plagiarized documents at top ranks.

Our aim in research question Q2 was to investigate alternative ways of document representation for indexing and retrieval. Experiments confirmed that the best performing IR approach is to extract terms from selective nodes of the AST of a source program, and then to store these terms in separate fields corresponding to the different AST node types from which they were extracted.

In research question Q3, we investigated ways of effectively representing a source code document as a pseudo-query. We find that best results are achieved when at most 20 terms are selected from each field of a source code document to constitute a pseudo-query for the retrieval phase.

For the supervised stage, we proposed three different categories of features, namely: lexical, structural and stylistic attributes. A random forest (RF) classifier was trained on the SOCO training set of plagiarized source code document pairs. Experiments confirm that a combination of these features is able to capture the inherent characteristics of plagiarism patterns by demonstrating that with the help of supervised classification, it is possible to further improve the results obtained from the IR stage. The implication is that a supervised classification approach is able to discard many false-positive cases form retrieved lists of candidate source codes thus improving plagiarism identification effectiveness.

Our future work will involve investigating more characteristic features for training the supervised approach to achieve further improvement in results; for instance, we may consider as an additional similarity matching strategy, the well known SimHash algorithm (Charikar 2002). For the retrieval stage, we plan to consider exploring the pros and cons of pseudo-relevance feedback for source code documents with an aim to further enhance retrieval effectiveness.

Acknowledgements The authors would like to thank to Enrique Flores, Paolo Rosso and Lidia Moreno for providing us with important details regarding the participating systems in the SOCO 2014 shared task. The first two authors are supported by Science Foundation Ireland (SFI) as a part of the ADAPT Centre at DCU (Grant No.: 13/RC/2106). The work of the last three authors was partially funded by CONACyT under the Thematic Networks program (Language Technologies Thematic Network Project No. 260178, 271622). Additionally, they would also like to thank to UAM Cuajimalpa and SNI-CONACyT for their support.

References

Baer, N., & Zeidman, R. (2012). Measuring whitespace pattern sequence as an indication of plagiarism. *Journal of Software Engineering and Applications*, *5*(4), 249–254.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings of the international conference on software maintenance, ICSM '98* (p. 368).

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.

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

Chae, D.-K., Ha, J., Kim, S.-W., Kang, B., & Im, E. G. (2013a). Software plagiarism detection: A graph-based approach. In *Proceedings of the 22nd ACM international conference on information and knowledge management, CIKM '13* (pp. 1577–1580).

Chae, D.-K., Kim, S.-W., Ha, J., Lee, S.-C., & Woo, G. (2013b). Software plagiarism detection via the static api call frequency birthmark. In *Proceedings of the 28th annual ACM symposium on applied computing, SAC'13* (pp. 1639–1643).

Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on theory of computing, STOC '02* (pp. 380–388). New York, NY, USA: ACM.

Cosma, G., & Joy, M. (2013). Evaluating the performance of lsa for source-code plagiarism detection. *Informatica*, *36*(4), 409–424.

Faidhi, J. A. W., & Robinson, S. K. (1987). An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers and Education*, *11*(1), 11–19.

Fernández-Delgado, M., Cernadas, E., Barro, S., & Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, *15*, 3133–3181.

Flores, E., Barrón-Cedeño, A., Rosso, P., & Moreno, L. (2011). Towards the detection of cross-language source code reuse. In *Proceedings of the 16th international conference on applications of natural language to information systems, NLDB 2011* (pp. 250–253).

Flores, E., Barrede, A., Moreno, L., & Rosso, P. (2014a). Uncovering source code reuse in large-scale academic environments. *Computer Applications in Engineering Education*, *23*, 383–390.

Flores, E., Rosso, P., Moreno, L., & Villatoro-Tello, E. (2014b). *PAN@FIRE: Overview of SOCO track on the detection of source code re-use*. In *Working notes of the forum for information retrieval evaluation, FIRE 2014*.

Flores, E., Rosso, P., Moreno, L., & Villatoro-Tello, E. (2014c). Pan@fire: Overview of soco track on the detection of source code re-use. In *Proceedings of the forum for information retrieval evaluation, FIRE 2014*.

Fox, E. A., Koushik, M. P., Shaw, J. A., Modlin, R., & Rao, D. (1992). Combining evidence from multiple searches. In *Proceedings of the first text REtrieval conference, TREC 1992, Gaithersburg, Maryland* (pp. 319–328), November 4–6, 1992.

Grieve, J. (2007). Quantitative authorship attribution: An evaluation of techniques. *Literary and Linguistic Computing*, *22*(3), 251–270.

Hiemstra, D. (2000). *Using language models for information retrieval*. Ph.D. thesis, CTIT, AE Enschede.

Jones, J. (2003). Abstract syntax tree implementation idioms. In *Proceedings of PLP '03*.

Kim, J. & Croft, W. B. (2012). A field relevance model for structured document retrieval. In *Proceedings of the 34th European conference on IR research, ECIR 2012* (pp. 97–108).

Marinescu, D., Baicoianu, A., & Dimitriu, S. (2012). Software for plagiarism detection in computer source code. In *Proceedings of the 7th international conference on virtual learning* (Vol. 156, pp. 373–379).

Narayanan, S., & Simi, S. (2012). Source code plagiarism detection and performance analysis using fingerprint based distance measure method. In *Procceedings of the 7th international conference on computer science and education, ICCSE '12* (pp. 1065–1068).

Neamtiu, I., Foster, J. S., & Hicks, M. (2005). Understanding source code evolution using abstract syntax tree matching. *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR'05*, *30*(4), 1–5.

Ogilvie, P., & Callan, J. (2003). Combining document representations for known-item search. In *Proceedings of the 26th annual international ACM SIGIR conference on research and development in information retrieval, SIGIR '03* (pp. 143–150). New York, NY, USA: ACM.

Ponte, J. M. (1998). *A language modeling approach to information retrieval*. Ph.D. thesis, University of Massachusetts.

Potthast, M., Hagen, M., Beyer, A., Busse, M., Tippmann, M., Rosso, P., & Stein, B. (2014). Overview of the 6th international competition on plagiarism detection. In *Working notes for CLEF 2014 conference* (pp. 845–876).

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

Sanderson, M., & Zobel, J. (2005). Information retrieval system evaluation: Effort, sensitivity, and reliability. In *Proceedings of the 28th annual international ACM SIGIR conference on research and development in information retrieva, SIGIR'05* (pp. 162–169). New York, NY, USA.

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (pp. 76–85). ACM.

Stein, B., Potthast, M., Rosso, P., Barredeo, A., Stamatatos, E., & Koppel, M. (2011). Fourth international workshop on uncovering plagiarism, authorship, and social software misuse. In *SIGIR Forum* (Vol. 45, pp. 45–48).

Takaki, T., Fujii, A., & Ishikawa, T. (2004). Associative document retrieval by query subtopic analysis and its application to invalidity patent search. In *Proceedings of the thirteenth ACM international conference on information and knowledge management, CIKM '04* (pp. 399–405).

Xue, X. & Croft, W. B. (2009). Automatic query generation for patent search. In *Proceedings of the 18th ACM conference on information and knowledge management, CIKM '09* (pp. 2037–2040). New York, NY, USA: ACM.