FULL PAPERS

USING CONCEPTS OF TEXT BASED PLAGIARISM DETECTION IN SOURCE CODE PLAGIARISM ANALYSIS

Michal Ďuračík, Emil Kršák, Patrik Hrkút

Abstract: At the present time the plagiarism is a growing problem due to a lot of easily accessible resources, and many papers deal with this topic. Some of them describe the plagiarism in written texts, others in source codes.

The goal of this paper, despite clear concept contrasts, is to point out the similarity of these worlds. We will analyse the potential of reusing known algorithms developed for textual corpuses in the area of source code analysis. It is obvious that the algorithms cannot be simple taken and reused without any changes, since a source code cannot be considered a plain text from the lexical point of view. The transformations performed to detect plagiarism in text documents differ from those suitable for source code processing. A promising approach to analyse source codes seems to be the creation of abstract syntax trees. The similarity comparison among linear structures is then transformed into tree-based structures similarities searching. However, nowadays such techniques are used only in mutual comparison of two files with source codes. This paper brings a review of different methods used in the source code analysis process. We are at the beginning of our research and we outline the steps which will be necessary to take with the source code analysis process (compared to plain written texts processing), and identify the possible issues primarily concerning plagiarism detection within a large dataset of files containing source codes.

Key words: plagiarism, source code, big datasets

1 Introduction

We live in a time of easy and, in our part of the world also unlimited, access to all sorts of information. The internet makes it relatively simple for students to access materials from all over the world, helping them with their studies. Not surprisingly, this encourages students to copy the said materials to be used in their work. However, the problem of plagiarism does not concern academic environment only. It has been an increasingly serious issue in different areas, from the field of education (Hammond, 2004) or journalism (Veselý, 2015) to patent infringement in the commercial sector (Holbrook, 2014).

Paradoxically, there are studies coming from academic environment itself pointing out a recent gradual reduction of plagiarism (Curtis and Kravjar, 2015). It is these studies which show the importance of plagiarism detection systems. The mentioned studies have reported a decline in plagiarism since plagiarism detection systems were integrated. As the systems are gradually being improved, students are getting

increasingly worried about being exposed for plagiarism. The fear has been acting as a counterbalance to how easy it is to commit plagiarism nowadays. As long as the systems are able to keep up with the current time, the trend may keep continuing.

The main reason the systems have been succeeding is the fact they are used globally. Every anti-plagiarism system verifies the authenticity of a piece of work being checked within a certain database. Usually, the systems create their own databases. Nevertheless, there are also systems that do not create their own databases but they use internet search engines instead (Williams, 2014). There are advantages to both ways, therefore, they are often combined. If a system uses just its own database, then it is not able to detect plagiarism of the works that its database does not contain. But, on the contrary, the system database may contain works that for various reasons cannot be made public and available through browsers. This primarily concerns scientific works, which are often sources of plagiarism, yet they are rarely freely accessible to public and internet browsers. Among the systems successfully using the above mentioned combination, we can list ANTIPLAG used in Slovakia, or the world-known system Turnitin.

The situation in the area of detecting source code plagiarism is different, though. Currently, there is no complex system or service allowing for the detection of plagiarism on the global scale. Therefore, no relevant statistics documenting the rate of plagiarism in the area are available. Over time, various algorithms have been developed, whether for text documents or source codes. Despite this progress, the system called measure of software similarity (MOSS) by the University of Stanford, and the system JPlag from Germany belong to the most widely used systems. Both systems were established more than ten years ago, and they have been being developed ever since. From the viewpoint of their design, they find their application particularly in the field of detecting plagiarism in various courses. Using them globally is practically impossible.

If we want to achieve the same trend for source code plagiarism as the one that has been shaping up for text documents, we need to start introducing systems that would check the originality of source code in a similar scale the systems for text documents do. Our long-term goal is to design such a system using the knowledge in text documents anti-plagiarism. We believe that in spite of the differences between text documents and source codes, these two worlds resemble each other, and the knowledge in one area will, without any doubt, find application in the second one, too.

2 How document processing works

To begin with, let us point out the difference between processing text documents and source codes for the needs of anti-plagiarism algorithms. We can divide processing a document into several steps:

- transferring the document into plain text
- tokenization
- removing stop-words
- stemming and lemmatization
- representation of the document

With systems that process text documents, the input documents are often documents in various formats (pdf, docx...). Except for the text itself, the documents contain a lot of additional information. Transferring the document into text is followed by tokenization. It is supposed to divide the text into individual tokens. Afterwards, so called stop-words are removed sometimes. Stop-words are frequently used words which, since they often occur in general, do not bring relevant information in the course of the analysis. The most common example of such word in English is the word 'the', which every document contains. The next steps are stemming and lemmatization. They are both supposed to unify various word forms, e. g. 'car', 'cars', 'car's', 'cars' into 'car'. Stemming is a more simple approach, looking for the roots of words by means of cropping. Lemmatization uses the morphological analysis to determine the word which the word being analysed has been derived from. The last step is the representation of the document, which various models are used for (bag of words, vector models), or the document is represented by means of n-grams.

Processing source codes is very similar. The main difference between a source code and text lies in their structures. We can divide text into chapters, paragraphs, sentences and words. Compared to that, a source code structure is considerably diverse. And, unlike text wherein the meanings are hidden in individual words, it is this very structure that defines the meaning of a said code. Processing source codes consists of the following steps:

- tokenization
- purification
- representation of the source code

Since a source code is usually written as pure text without any formatting, in contrast to text documents no primary processing is necessary. Instead, tokenization is carried out immediately. Source code tokenization is more demanding than text tokenization. For simplicity, with text processing the term 'token' is understood as 'a word '. Therefore, in text tokenization the text gets divided into tokens and separators. With source codes, however, not individual words but sequences of characters with certain grammatical meanings within the given language are considered as tokens. Moreover, the tokenization output is not a list of the sequences. The sequences are replaced by relevant identifiers, on the basis of their meaning within the grammar of the language. For example, the expression `a = 5` may be processed into the tokens: *variable, assignment_operator, constant*. After tokenization, the source code needs to be represented in a particular way. Since we are dealing with tokens, it is possible to use the representations as they are used with text documents (bag of words, vector model...). But, as we have mentioned already, a large proportion of information on a source code is hidden in its structure. With the already mentioned representations the information gets lost. Not only is the structure important for the needs of analysis but it is used by programming languages themselves. In compiling, some programming languages made do with these tokens, which they more or less had translated 1:1 into an executable code. In the course of time, as programming languages were developing and their grammar was becoming more and more complicated, new ways of source code representation emerged. The basic one among them is syntactic tree. A syntactic

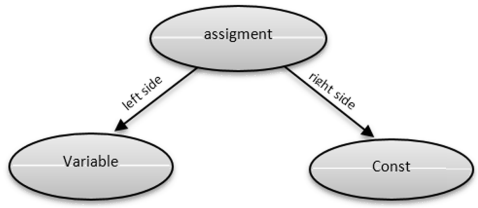

Figure 1. Syntax tree

tree is just tokens arranged into a tree, on the grounds of their structure derived from the language grammar. Figure 1 presents our example with the expression `a = 5` shown in a simplified syntactic tree.

Very often the trees are used for program modification of source code, e. g. in development environments. This is mainly because they perfectly describe what is written in the source code. But if we need to analyse only the meaning of source code, and we do not care what syntax has been used to write it, then syntactic trees will not make our work easier. That is why, in addition to syntactic trees, also abstract syntactic trees (AST) are used. Like the name itself suggests, they are abstract, they describe the meaning of a given code independently on particular syntax. ASTs are usually constructed of syntactic trees. The main difference between a syntactic tree and an AST is that the syntactic tree describes syntax, and the AST describes the logical scheme of a program. Our simple expression `a = 5` would not show us the difference between the syntactic tree and AST because it contains only the simple allocation operation. Let us present a different example – the expression `sum(1,2)`. Let the expression represent a function call called `sum` and having two parameters.

Figure 2 shows the difference between a syntactic tree and an AST. The syntactic tree of this expression has two parts – an identifier and a list of parameters. The AST, to the contrary, represents the expression by means of the node function call and its relevant parameters. The AST is more simple, and it contains only relevant information on the source code logical structure. Its further advantage is also the fact that it is independent on the used programming language.

3 Document similarity measurement

So far, we have been describing only the way of processing documents, or source codes, respectively. To look for similarities among documents, one of the already mentioned representations is used as the basis. On the grounds of the representation being used, we can divide the methods into two main groups. In the first one, there are methods that use one of the statistical representation of the document (bag of words, vector model...). The advantage of these methods is that they work with a document representation which is usually considerably smaller than the original document is. The disadvantage is that the result of the comparison is only 'a number ' that determines the distance among documents (Euclidean distance, Jaccard Coefficient...) (Huang, 2008). In the second group, we find the methods working with full document

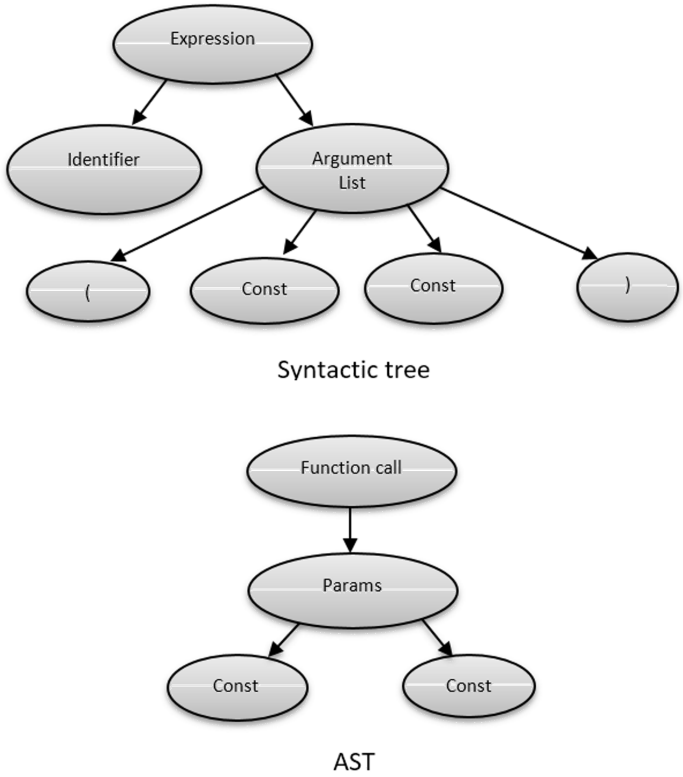

Figure 2. Difference between syntax tree and AST

representation (some n-grams based methods, syntactic trees). Their main advantage is that they can find particular matching parts. On the other hand, their disadvantage is being usually more demanding on memory and computation.

When anti-plagiarism systems are created, both groups get combined. First, by using a method from the first group, documents are clustered. The clustering usually reduces the number of documents to compare. Then a method from the second group is used to find particular similarities among the documents.

3.1 Common methods

There are a lot of works dealing with plagiarism and examining similarities in text documents or source codes. We can find descriptions of the methods used to determine documents similarities in the work *A Survey of Text Similarity Approaches* (Gomaa, 2013). In addition to the standard methods working with tokens, the authors deal with other methods, too. However, the methods do not make much sense in the context of source code because they use various corpuses designed for language research. With source code, there are also a few methods that are used. We can find their summary

and descriptions in the work *Source Code Plagiarism Detection 'SCPDet': A Review* (Gondaliya, 2013). Except for the individual methods, its authors describe also ready-to-use tools to detect plagiarism in source codes.

In the following part we describe some methods from each of the groups. We examine clustering and document matching methods, since every one of them possesses certain specific attributes.

3.2 Clustering

Classification algorithms used for text documents are not possible to apply directly to source code because the tokenization of text documents produces a diverse list of tokens (terms). With source code, the list is significantly truncated, since programming languages usually have limited numbers of tokens. Each larger piece of source code will contain almost all tokens available in the programming language. In addition to the standard source code tokenization, it is also possible to extract identifiers from the source code and to do the relevant analysis on them. Again, extracting the identifiers correctly and subsequently processing them is not simple, however, there have already been methods to do that (Carvalh, 2015).

The basic methods of documents clustering on the grounds of their mutual distances are also used in the case of text documents (Huang, 2008) in source codes (de Hoo, 2004), too. The mentioned works show that using euclidian distance is possible with texts, as well as with source codes. Fuzzy-based methods might be another example: they have been showing their potential with both, text documents (Trappey, 2009) and also source codes (Acampora, 2015). Currently, numbers of documents compared are remarkably large, and most methods come up against memory limits. Therefore, methods based on Latent dirichlet allocation (LDA) (Blei, 2003) have been used for text clustering. The same method has been proved to be applicable also for source code (Binkley, 2014), and, besides classification, it has been applied to other areas, too (Thomas, 2014).

3.3 Document matching

When finding identical parts, in the case of text documents we work with similarities on the level of characters or tokens. With source code it is with tokens or some of the already mentioned tree representations. The most known method used also with both, text documents (Noynaert, 2006) and source codes (Prechlet, 2000), is Running Karp-Rabin Matching and Greedy String Tiling (RKR-GST). It is based on searching for the longest common sub-string in both documents. For text documents, it uses document representation by means of individual characters. With source code, the code is represented by a string of tokens.

Another frequently used method, whether with text documents (Schleimer, 2004) or source codes (Bahmani, 2010), is winnowing. It uses so-called local fingerprinting. The basis of its algorithm is document representation by means of n-grams. The n-grams are subsequently used for the calculation of document fingerprint. Finally, similarities among the fingerprints are looked for.

The last group of methods we are going to mention are methods based on tree structures comparisons. They do not occur with text documents, however, with source code they are quite common. Although they can also use syntactic trees, it is AST they use most frequently. To compare tree structures, hashing is used. A hash is calculated for each AST node. The hash depends on the node itself, as well as on its descendants. This way we get a tree in which every node is able to describe its entire sub-tree by one value. When seeking similarities, we look among individual AST for nodes with similar hashes. The advantage of this approach is primarily the fact that a hash calculation may be adjusted to be able to detect commonly used transformations (Tao, 2013).

4 Summary

As we have demonstrated above, the interconnection between the world of text documents and the world of source code is possible to be achieved successfully. A lot of algorithms utilized in the two individual worlds share a common base. What they differ in is particularly the way they process the input file. The analysis that follows may be even identical. This, however, does not mean that using an algorithm created for text will achieve equally good results with source code, too.

Despite that, currently there is not a system possible to be used to analyse source code plagiarism on a larger scale. On the other hand, there are a few such systems for texts. Algorithms that would be able to do that for source code are known (Burrows, 2007). The problem is no system has been using them so far. In addition, we also consider their principle as problematic, since they use methods utilized with text documents. Nevertheless, the latest researches indicate that AST has been more efficient in this respect (Tao, 2013).

How to build a system for global detection of source code plagiarism

There are several possible directions to take when creating such a system. The first of them is building the system on the basis of already existing solutions, which would make the whole problem more simple. The second one is designing a new algorithm which would use the latest knowledge of source code plagiarism detection. Using AST as the basis for such algorithm has a few advantages. The first of them is, of course, efficiency. Then they are further possibilities of using the algorithms. One of them might be also effective detection of structures within large datasets.

On the grounds of the current knowledge we know that such system needs to consist of several parts:

- input data processing
- indexation
- similarities detection
- resulting similarity calculation

We consider the four parts as necessary for the system to work correctly and efficiently. Dealing with input data processing does not make much sense any more. Currently, there are plenty of tools[1] that can extract tokens from source codes or

[1]ANTLR – http://www.antlr.org

create AST directly. The process we have described in the part 'How document processing works' includes all important steps of the input source code processing: source code purifying – removing comments, spaces. Tokenization provides source code standardisation. Subsequently, the standardized code is transformed into AST form. An interesting issue is the identification of irrelevant code parts. Each source code contains certain parts that we can find in almost all other source codes. With text documents, the problem is solved by means of stop-words. A similar principle will have to be applied also with source codes.

Another step to take will be the indexation of the already processed source code. Tokens indexation algorithms exist, and they seem to be efficient enough (Burrows, 2007). As far as tree structures indexation is concerned, it might be a bit more complicated. There are various document databases, graph and object databases. Nevertheless, the question remains if they could be used for the case at all, and, if so, to what extent. Current algorithms using AST, which we have described in this article, use tree hashing and not direct comparisons of two tree structures. Therefore, the index will not necessarily have to describe a tree structure but it will be possible to fragment it into a relational structure and to use some of the relational databases.

Besides the indexation, how to use the index for searching is equally important. The way has to be based on the indexation being used, with the goal of using the index as efficiently as possible. With detecting similarities within a large dataset, pre-processing is also important. By pre-processing we mean reducing the amount of work aimed at the verification of source code. For pre-processing it is possible to use some of clustering methods (Aggarwal, 2014) which help us reduce the set being searched relatively quickly.

The last part is the final calculation of similarities among the detected works. Again, there are several ways to calculate similarities between two source codes, and it is crucial to select the right one. The similarity coefficient should express the percentage rate of the coincidence among the individual works. Last but not least, we also have to consider the modifications carried out. Even though they do not change the meaning of the code, they change the entry form. Therefore, the calculation should take them into consideration to a certain extent.

5 Future work

We believe the basis of plagiarism detection systems within source codes is not in currently used algorithms but in algorithms based on AST, which have shown their advantages quite clearly. Nowadays, comparing documents representations by means of tokens has been managed very effectively. Despite that, we have decided to work with AST. Current algorithms will not be of much help because they are not suitable for working with tree structures. That is why we would like to keep examining the possibilities of tree structures indexation, and subsequent searching in the indexes. We are confident it is the right path source code plagiarism detection should be taking. In addition to anti-plagiarism systems, we believe the method can be also used for the analysis of vast transportation systems, such as KANGO, GTN, VIS (Tavač, 2014). The

systems share a lot of common elements, and analysing them might help us extend them further

Literature

Acampora, G., Cosma, G. (2015): A Fuzzy-based approach to programming language independent source-code plagiarism detection. In *Fuzzy Systems* (FUZZ-IEEE), 2015 IEEE International Conference on (pp. 1–8). IEEE.

Aggarwal, C. C., Reddy, C. K. (Eds.). (2013): *Data clustering: algorithms and applications.* Chapman and Hall/CRC.

Bahmani, Z., Taleb, N. (2010): *Fingerprinting Jar Files Using Winnowing and K-grams.*

Blei, D. M., Ng, A. Y., Jordan, M. I. (2003): Latent dirichlet allocation. *Journal of machine Learning research,* 3(Jan), 993–1022.

Burrows, S., Tahaghoghi, S. M., Zobel, J. (2007): Efficient plagiarism detection for large code repositories. *Software-Practice and Experience,* 37(2), 151–176.

Carvalho, N. R., Almeida, J. J., Henriques, P. R., Varanda, M. J. (2015): From source code identifiers to natural language terms. *Journal of Systems and Software,* 100, 117–128.

Curtis, G. J., Vardanega, L. (2016): Is plagiarism changing over time? A 10-year time-lag study with three points of measurement. *Higher Education Research & Development,* 35(6), 1167–1179.

de Hoon, M. J., Imoto, S., Nolan, J., Miyano, S. (2004): Open source clustering software. *Bioinformatics,* 20(9), 1453–1454.

Gomaa, W. H., Fahmy, A. A. (2013): A survey of text similarity approaches. *International Journal of Computer Applications,* 68(13).

Gondaliya, T. P., Joshi, H. D., Joshi, H. (2014): Source Code Plagiarism Detection'SCPDet': A Review. *International Journal of Computer Applications,* 105(17).

Hammond, M. (2004): Cyber plagiarism: Are FE Students Getting Away with Words. *Plagiarism: Prevention, Practice and Policies Conference* in St. James

Holbrook, T. R., Osborn, L. (2014): *Digital patent infringement in an era of 3D printing.*

Huang, A. (2008): Similarity measures for text document clustering. In *Proceedings of the sixth new Zealand computer science research student conference* (NZCSRSC2008), Christchurch, New Zealand (pp. 49–56).

Kravjar J. (2015) SK ANTIPLAG Is Bearing Fruit. *Plagiarism across Europe and Beyond 2015 – Conference Proceedings,* pp. 147–163.

Noynaert, J. E. (2006): Plagiarism detection software. In *Midwest Instruction and Computing Symposium.*

Prechlet, L., Malpohl, G., Philippsen, M. (2000): *JPlag: Finding plagiarisms among a set of programs.* University Karlsruhe.

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.

Tao, G., Guowei, D., Hu, Q., Baojiang, C. (2013): Improved plagiarism detection algorithm based on abstract syntax tree. In *Emerging Intelligent Data and Web Technologies* (EIDWT), 2013 Fourth International Conference on (pp. 714–719). IEEE.

Tavač, M., Bachratý, H. (2005): VIS – nový informačný systém pre vyhľadávanie spojení v dopravných sieťach. In: *Infotrans 2005 informační technologie v dopravě a logistice*, IV. Ročník mezinárodní

konference, Pardubice, 21.–22.9.2005, s. 289–297 elektronický zdroj Pardubice Univerzita Pardubice 2005, ISBN 80-7194-792-X.

Thomas, S. W., Adams, B., Hassan, A. E., Blostein, D. (2014): Studying software evolution using topic models. *Science of Computer Programming*, *80*, 457–479.

Trappey, A. J., Trappey, C. V., Hsu, F. C., Hsiao, D. W. (2009): A fuzzy ontological knowledge document clustering methodology. *IEEE Transactions on Systems, Man, and Cybernetics,* Part B (Cybernetics), 39(3), 806–814.

Veselý, O. (2015): Results of similarity analysis of online news in Czech Republic. *Plagiarism across Europe and Beyond 2015 – Conference Proceedings,* pp. 99–106.

Williams, K., Chen, H. H., Giles, C. L. (2014): Classifying and ranking search engine results as potential sources of plagiarism. In *Proceedings of the 2014 ACM symposium on Document engineering* (pp. 97–106). ACM.

Copyright statement

Authors

Michal Ďuračík (michal.duracik@fri.uniza.sk), Emil Kršák (emil.Krsak@fri.uniza.sk), Patrik Hrkút (patrik.hrkut@fri.uniza.sk), Faculty of Management Science and Informatics, University of Zilina, Univerzitná 8215/1, 010 26 Zilina Slovakia