Automated Tools for Source Code Plagiarism Detection

Kristian Wahlroos

Seminar report
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, April 19, 2017

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Kristian Wahlroos

Työn nimi — Arbetets titel — Title

Automated Tools for Source Code Plagiarism Detection

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminar report	April 19, 2017	11

Tiivistelmä — Referat — Abstract

Plagiarism is a serious problem which can be hard to detect. Especially in massive online courses (MOOC), where students are required to complete programming tasks, there is no other guarantee than trust, that the student has really completed his/her assignment completely by themselves. Same kind of situation is possible in academic world, where universities held courses that relies on students completing various programming tasks. This setting can lead to cases of plagiarism where the solution for a given exercise is actually copied from a study friend or from the Internet. Also obfuscation is often used by plagiarists to hide their cases of plagiarism, which makes detecting plagiarism even harder and with the sheer number of participant, the plagiarism detection is almost impossible to do as a manual labor.

This paper describes the current solutions for automated plagiarism detection from the source code by doing a literature review. The focus will be on the machine learning aspect and the scenario that plagiarism is reflected to in the end of this paper, is a scenario where student is completing programming assignments for a grade. This kind of scenario is very common one among universities, and exists also at University of Helsinki's course '*Introduction to programming*'.

Avainsanat — Nyckelord — Keywords
Machine learning, Source code, Plagiarism, Author identification

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Massive online courses or *MOOCs* have gained a lot of popularity in recent years. Their course formats are easy to follow and don't require any live attendance from the students to be able to complete the courses. MOOCs also has a quite flexible schedules to complete them, which makes them very interesting in the eyes of a student. Some MOOCs like edX[1] and Coursera can even offer real credits for the students, if students universities have made a deal with them. From available MOOCs, programming has gained a lot of interest and in the end of 2016, edX even listed three programming courses as the years most popular courses[2]. The course formats of programming MOOCs work more or less the same way; videos for lectures and returnable programming assignments as tasks. These programming assignments are usually automatically tested and credited for the student, and this makes it very easy for the student to complete programming courses from their home computers whenever they want.

Many MOOCs work by trusting that the student has completed exercises by themselves and this is not really guaranteed in any way. In other words there usually doesn't exists any automatic system for possible plagiarism detection inside these courses. This leaves MOOCs in a rather difficult position, because they offer credits based on one-way trust, and without a final exam with mandatory attendance, one-way trust becomes a quite shallow. For example, source code plagiarism is relatively easy and without automatic tools it's almost impossible to get caught if there are hundreds or even thousands of students. Detecting source code plagiarism is thus basically impossible for the course keepers without any automated tools.

In this paper tools and methodologies for automated source code plagiarism detection is reviewed. The scope will be focusing on various machine learning techniques, while the reflected scenario is kept to be at MOOC-style courses or academic world where plagiarism could happen within the normal programming course with assignments. Also the term *machine learning technique* is considered as a term, that covers following subcategories: unsupervised learning, supervised learning, data mining and various probabilistic models. Thus it's not limited only to analyze for example labeled data, meaning already known cases of plagiarism in one scenario.

The structure of the rest of this paper will be following: starting with the background motivation follows the methodology part where used technique for this literature review is described; then comes the results where machine learning techniques and features from various papers are presented; analyzing part sums up those findings to find common cases or really differing techniques; finally comes discussion part that reflects the findings on how could these be

[1] See more at `https://www.edx.org/credit`

[2] Based on `http://blog.edx.org/10-most-popular-courses-edx-courses-in-2016`

used at programming related MOOCs.

1.1 Background

As there are countless of opportunities in MOOCs to plagiarise source code, there are as equal amount of ways to do it in academic courses. Usually plagiarism in courses is not necessary even intentional but just by-product of students doing the same tasks together. For example in *Introduction to programming*[3] course organized by University of Helsinki, the course lasts 7 weeks and every week student has to complete programming assignments with Java-language. Every student has same assignments in the same order and they are encouraged to help each others during sessions. This naturally can create problematic cases of group work, which as recurring events are also real plagiarism in a form of source code sharing. One solution in here would be automatically detect those cases and then try to resolve them manually.

Other types of source code plagiarism are direct plagiarism and obfuscated plagiarism. Direct plagiarism in source codes is basically copy-pasting the code and it's the most simplest form of plagiarism. Obfuscation means that the plagiarist is trying to hide his/her plagiarism by making the code look more like it was made by him/her. In source code, obfuscation could relate to renaming variables and functions or making slight modifications into the plagiarised code for example by changing the order or making slight modification into the logic of the code.

2 Methodology

The references for this paper were gathered by performing systematic literature review utilizing *Google Scholar*. This was done in a two step manner where first a collection of papers were gathered by searching with specific keywords, then these papers were filtered if their abstract, keywords, title or introduction contains specific terms.

In the first step, used search terms which required a direct match were **machine learning**, **plagiarism** and **code**. Other terms used were **authorship** and **identification** but these didn't require a direct match from the results. Papers were filtered also by year, only considering papers from 2006 onward. This was done because most modern MOOCs like Coursera and Udacity are relatively new, and automated programming assignments haven't existed before. Papers from 2006 onward also have a high probability to consider modern programming languages for example Java or Python, which both are heavily used to teach programming in universities[4].

[3]This years course page at `https://2017-ohjelmointi.github.io/`

[4]Based on `http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext`

Second step contained a more specific filtering, where only papers that included words **plagiarism**, **programming** or **code** were selected. This was done to get more specific results which focuses more on the scope of this paper. That is, detecting source code plagiarism with machine learning.

Gathered papers are analyzed together to form the final results of this paper. This is done by looking at the methodologies and results of gathered studies by their accuracies, features extractions and usage of machine learning techniques. This is believed to help to give answers on how plagiarism could be detected from the source code, targeting especially academic courses and MOOCs.

3 Results

Machine learning in a source code plagiarism detection works like any other machine learning classification technique in another scenario. It is used to find recurring patterns from the data that could be used to separate it into individual parts. In source code plagiarism, this means finding features that could be used to identify the author and classify unseen data to most likely author. Unseen data in this sense means the source code from an unknown author, and the goal is to predict that which one of the possible authors wrote the unseen source code or is the unseen source code too similar to some other code.

Previous scenario is issued in [1], where Bandara and Wijayarathna are using attribute counting technique to develop a machine learning model to detect source code plagiarism. Attribute counting technique means, that the original source code is transformed into feature vector that contains only stylish metrics i.e. no data from the underlying structure. They report final accuracy as 86.65%. Same idea of using style metrics from source code was researched earlier by Lange and Mancoridis [6]. They use 18 various features and measure distribution differences of the metrics. Lange and Mancoridis report the final accuracy of their model as 75%. Also Kothari *et al.* take the coding style and character sequences as their features for the model [5]. They have six different style and text distribution metrics which can represent the distribution of n-character patterns[5]. Their reported accuracy when using students source codes is for the pure style metrics 36% and for the n-character 69%. Elenbogen and Seliya build a data mining model based on six different style metrics in [3]. They call these as a *programming profiles* and these style metrics are described as elements that doesn't affect the functionality. Their accuracy using 83 programs and 12 students was 74.7%.

Other studies use more structural analysis of the source code to form the features. In [4], Jadalla and Elnagar describe the *PDE4Java* detection engine to detect plagiarized Java-code. Authors use tokenization to build

[5]For example using 2 character patterns slice the source code into two character slices.

the needed format from the source code, that can be utilized for the N-Gram representation. This N-Gram representation is basically same as splitting the source code into slices of chosen length. Their data mining method reported suspicious source codes from eight various sized data sets about 6 out of 8 times in the same way as domain expert reported. The authors however didn't include the specific accuracy of the data mining model. Comparing to the study of Jadalla and Elnagar, even more structural-based analysis is done by Caliskan-Islam *et al.* They use various code stylometries derived from abstract syntax trees to build the model that can de-anonymize programmers [2]. Their model accuracy is 94% when there were 1 600 possible authors and 98% when there were 250 authors. Rosenblum *et al.* researches that does programming style preserve after compilation [7]. They use style metrics for authorship identification, but this time style metrics from the compiled code as original source code might not always be available, making the style metrics more structural based. Their validation used 20 authors and the model reaches 77% accuracy. Finally, Son *et al.* propose a parse tree kernel to form the model based on structural features in [8] using 555 source codes. Their reported accuracy was 93% same as using a human validator for the same task.

The collected papers seems to divide into two different categories: pure stylistic features [1, 6, 5, 3] and structural based features [4, 2, 7, 8]. As most stylistic metrics are based on easily collectable metrics from the original source code, structural approaches usually performs pre-processing to get the underlying features. Pre-processing is in many cases done by forming abstract syntax tree -representation which is not affected for example if names of the variables are changed.

3.1 Data

The data sets used and their sizes in previous studies vary a little bit; source codes from coureses, synthetic data sets, codes collected from open source projects and codes from open competitions for example *Google Codejam*. Most of the variance comes from the number of possible authors and the number of used source codes, but based on collected papers, one can't say does it matter if the method structural or style based. The findings related to data set sizes are summed into following table, where *size* refers to the total amount of files used, and *authors* as the total number of authors reported in data set.

Attr./Paper	[1]	[5]	[3]	[6]	[8]	[2]	[4]	[7]
Size	741	200	83	4068	555	N/A	326	203
Authors	10	8	12	20	N/A	1600	N/A	32

Table 1: Reported data sets used in papers.

In table 1, values are chosen only if they are explicitly told in the paper. If there have been multiple values or data sets, data set size have been selected as the one with student-related data, which is often data collected from various courses. In three studies explicit values are not told and that is why they are marked as *N/A*. For example in the study of Caliskan-Islam *et al.* [2], their data set was collected from Google Codejam which explains their large amount of programmers. However they didn't explicitly tell how many source code files there were in total, but rather mentioned that there were around one to 17 code files per a contestant. In two other cases number of explicit authors remained untold and all the missing valued papers were from structural analysis.

3.2 Methodologies

In this section, methodologies of collected studies are summarized, but features and feature extraction is left out for the next section. Methodology is viewed here as the common concept in machine learning which is usually characterized by choosing the machine learning method for example *logistic regression*, for classification. It seems that from gathered papers, many style-related studies seems have similar features, but their machine learning techniques differ in some sense. In structural-related, their methodologies can be very different but often utilizing the gained information from the abstract syntax tree. Also data mining seems to be used and tree-related algorithms in structural studies.

The overview from these used techniques can be summarized into following table.

Attr./Paper	[1]	[5]	[3]	[6]	[8]	[2]	[4]	[7]
Method	E_3	NB/VFI	DT	GA	PTK	RF	DM	K-M/SVM

Table 2: Methods used in papers. Abbreviations are listed on below.

$\mathbf{E_i}$	Direct ensemble of i models
NB	Naive Bayes
VFI	Voting Feature Interval
DT	Decision Tree
GA	Genetic Algorithm
PTK	Parse Tree Kernel
RF	Random Forest[6]
DM	Data Mining (DBSCAN)[7]
K-M	K-means Clustering
SVM	Support Vector Machine

[6]Could also be classified as ensemble method but for clarity it's separated here

[7]Paper describing DBSCAN `http://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf`

The only ensemble method from table 2 is used in [1] and it consists of k-nearest neighbors, Naive Bayes and Ada boosting algorithm. Authors say that they use Ada because it improves the accuracy of their model, and it helps to deal with weak learning algorithms. Kothari *et al.* in [5] are comparing side to side Naive Bayes and Voting Feature Interval. It was measured that VFI provided better results from open source projects and Naive Bayes from student based source codes. Another study that used two various algorithms for prediction is by Rosenblum *et al.* in [7]. They used SVM for authorship identification and k-means to group stylistically similar programs together. Authorship identification was used for finding the most likely author based on source codes stylistic features. This model was then utilized when the clustering was done to group the most similar source codes together. Clustering method is also studied in [4] where Jadalla and Elnagar present their plagiarism detection system *PDE4Java*. Their clustering method is a density based clustering (*DBSCAN*) which is controllable by two different variables: minimum number of clustering forming points and radius around a centroid.

Other classification algorithms used in papers are decision trees and genetic algorithm. Trees are used for structural-based classification in [8, 2] and for stylistic-based in [3]. Structural-based studies utilize heavily the syntax tree as first parsing the code and then using the syntax tree to form similarity measures that will distinguish source codes from each others. This is because syntax tree and tree-based models are both tree-like structures, so it's natural to take advantage of this property. However, parse tree kernel and random forests differ from each others as parse tree kernel is used to calculate pair-wise similarities between source codes, random-forest is used to train multiple decision trees, which have to be first trained in order to make predictions. This viewpoint differs a lot from parse tree kernel that utilizes just the similarity matrix and is not so predictive-based solution. In stylistic-study decision tree is utilized, according to the authors, because it allows a direct view to the final model that allows to see how various features affects to it.

Genetic algorithm is only used in study of Lange and Mancoridis in [6], and it's the oldest study together with the study by Kothari *et al.* in [5] from the year 2007. Genetic algorithm differs a lot from other methods because rather than a direct predictive model based on statistics for example Naive Bayes, it is utilized first to form the best feature combinations and then to classify the data. As a fitness function, they used the correctly classified documents i.e the precision to reward the algorithm.

It can be seen from studies that their methods divide roughly into four different categories: probabilistic models, tree-learners, clustering-based models and evolutionary algorithm. Interestingly studies that utilize the information about the source code structure use either clustering or tree-

based methods, as in style-based methods there are more variance in the model selection.

3.3 Feature extractions

To gain even more insight to the used machine learning related methods, feature extraction methods needs to be inspected from the papers, as this is also the topic that differs them from each others the most. In this context, feature extraction is considered as the method to change the source code document into vector of either continuous and/or discrete numbers that will represent the original document. In the case of supervised learning problems where documents are labeled by authors, this representation can be formalized usually into following equation $\mathbf{X} = (x_1, x_2, ..., x_n, y)$. In this equation $\mathbf{X}$ represents the source code, x is the individual feature used and y is the output variable which is from the plagiarism perspective, usually the author of the document.

In this paper, it has been shown that machine learning methods used to find plagiarism in source codes divide into two main categories: stylistic and structural studies. This division is next utilized with the findings from the subsection 3.2 to inspect feature extraction from various viewpoints.

Stylistic features Usages of stylistic features are clearly found in [1], where Bandara *et al.* count nine different metrics that they used. Some of these metrics are gained by parsing the source code with ANTLR[8] parser generator but most of them can be gained directly from the code. Bandera *et al.* represent the original source code as a set of tokens as for example number of characters per line, number of words in document and relative frequency of access modifiers[9] used. The frequencies of these nine tokens are then calculated and gained metrics are directly used as an input for the learning algorithm.

Similar approach, but from the pure probabilistic perspective, is given in [5], where Kothari *et al.* uses a three step pipeline to transform source codes into features. First they extract metrics via extraction tool, then filter out non-meaningful metrics and finally form a profile database which captures the writer profiles. For the actual metrics, they use two different kinds of metrics: style and pattern based. Style based metrics are very similar to the ones used by Bandara *et al.*, consisting of for example usage of line sizes, leading spaces and commas per line. The difference is that Kothari *et al.* forms a distribution out of stylistic metrics i.e. grouping values of metrics by intervals which is done because of the probabilistic nature of their model. Pattern based features are constructed by forming a distribution from the source code out of all possible character sequences length of n. For example

[8]More information about ANTLR parser at `http://www.antlr.org/`

[9]Access modifiers in Java are `public`, `protected` and `private`

considering statement "$y␣=␣5$", where character "␣" denotes the space, the 2 character sequences are (y␣), (␣=), (= ␣) and (␣5). Authors say that the 4 character sequences provided the best results.

Elenbogen and Seliya refers also to building programmer profiles in [3]. They utilize data mining tool called *Weka* which is used to parse the source code documents. As features, they use six different code metrics: zipped size, lines of code, number of variables, number of for loops, number of comments and variable length. Authors also tell that in their study, the most meaningful metrics to distinguish authors are lines of code, number of comments and variable length.

The final stylistic study is by Lange and Mancoridis [6]. They utilize 18 different base metrics and via genetic algorithm, find good metric combinations. After extracting metrics from the source code, histogram distributions are built to differentiate developers based on distribution differences. Inside these histograms values are normalized in a way, that they sum up to one because according to authors it is necessity, if one wants to compare different sized code masses. As for the actual metrics, they use text-based and syntactic metrics. Text-based are gathered directly from the source code without any need of external tools and syntactic measures require a slight parsing. Some text-based metrics they use are length of lines, numbers of statements per line and numbers of periods in identifiers. Correspondingly few of the syntactic measures are access methods scopes, control flows and switch-case structures. From these 18 metrics the best performing in the final validation were metrics related to usage of curly braces, comments, indentation, inline spaces/tabs, length of lines, periods, switch-cases, usage of underscore and the frequencies of first characters used in identifiers.

Structural features Structural studies takes a slightly different point of view into the features. For example in [8], Son *et al.* utilize parse tree kernel as their base model. Using parse tree enables to detect plagiarism disregarding the stylistic metrics which can be easy to obfuscate. The idea behind parse tree is to create tree representation from the source code and utilize properties of that tree to count how similar trees are. The steps required to create classification are thus first extracting the tree, generating similarity matrix and finally form a group of most similar documents. For the parse tree generation, authors use *ANTLR* to easily generate parse tree from the source code. After the parse tree has been generated, modified parse tree kernel is used to compare tree-like data structures i.e. form similarity matrix. This modified tree kernel differs from traditional text-based kernel in two ways: the depth has to be controlled as source code creates very deep nesting and the order of nodes is different when compared to traditional written text. For a similarity measure, authors use a measure called *maximum node value*, which finds the maximum similarity rather than the other option, which

would be the mean.

Tree-structure is also utilized for features in the study by Caliskan-Islam *et al.* in [2]. They form *Code Stylometry Feature Set* which is collected directly from the abstract syntax tree (AST) representation of the source code and as complete, contains roughly 120 000 different features. This feature set divides into three parts: lexical, layout and syntactic features. Lexical and layout features are really close to features seen in stylistic features consisting for example number of functions, number of keywords, average length of lines and indentation frequencies. Syntactic features on the other hand, are utilizing directly abstract syntax tree generated for each function. Some of the features based on syntactic elements are maximum depth of AST node, AST node 2-gram term frequencies and average depth of possible AST nodes. Authors also tell that bigrams discriminated the most and that they also utilize besides term frequency, the inverse document frequency as a measure to find rare occurrences of various keywords. The problem however was, that 120 000 features were too much and the feature space needed pruning, thus pruning was done by performing feature selection by information gain -measure. The final feature space consists couple hundred of non-sparse features.

Approach differing from two previous ones, is taken in [4] by Jadalla and Elnagar. They transform the source code into tokens or n-gram units via tokenization. Tokenization happens by *JLex* and *Java CUP*. Together these tools transform the source code into tokenized representation, which contains placeholders for example for keywords, separators and identifiers, instead of the actual written code. After tokenization, output is used to build n-gram model using 4 as the value of n. This N-gram works as a input for inverted index structure which, according to authors, allows a fast search to see if a same n-gram exists in multiple documents. Finally a similarity measure is needed for clustering, which in their study is selected as *Jaccard coefficient*. Jaccard coefficient is defined between two documents as the ratio between total frequency of a specific n-gram and the sum of total frequency, frequency in document A but not B and vise versa. It's notable that they don't use any other features for classification. This is mostly because their clustering approach can be constructed only by utilizing values of document similarities.

Lastly, a study by Rosenblum *et al.* uses graphs and n-grams as features, when they are doing authorship classification from binary code [7]. To gain the necessary features from binary presentation, they parse the binary code to machine language. This parsed document is then used as a base for creating control flow graphs and n-grams. Control flow graphs are built from four different models: idioms which are short instruction sequences, graphlets which reflect the local structure of the program by coloring function, supergraphlets that are merged graphlets and finally call graphlets which capture the interaction with external libraries and internal system level

instructions. N-grams, on the other hand, are byte string length of 4 designed to capture small nuances of the style that particular programmer might follow.

4 Analysis

As literature review here has shown, there are quite a few ways to find plagiarism via machine learning but these methodologies can be easily grouped into two: stylistic attribute counting methods and structural methods. As attribute counting mainly focuses on countable metrics which can be extracted easily from the given source code, structural studies often can outperform those by using a tree-like representation from the source code that is immune to trivial obfuscation tries. This tree-like structure is often parsed directly from the source code and is often in a form of abstract syntax tree. However, stylistic features are shown to be important to capture a specific style of the programmer as that information is often impossible to find from the parsed source code. Popular methods on attribute studies have been for example counting indentation, access methods, variable usage, n-grams, length of programs and length of lines. On structural, there have been for example sizes of syntax trees, number of branches, similarity measures and measures from control flow diagrams. If inspecting the data studies have used,

For the course in plagiarism

5 Discussion

TODO: Main points/main ideas. Limitations

6 Conclusion

As we have seen in this paper, many plagiarism detection studies are also focused into authorship identification. This is because plagiarism can be thought as a special case of authorship identification, where the misclassification actually is more interesting case.

References

[1] Bandara, Upul and Wijayarathna, Gamini: *A machine learning based tool for source code plagiarism detection.* International Journal of Machine Learning and Computing, 1(4):337, 2011.

[2] Caliskan-Islam, Aylin, Harang, Richard, Liu, Andrew, Narayanan, Arvind, Voss, Clare, Yamaguchi, Fabian, and Greenstadt, Rachel: *De-anonymizing programmers via code stylometry.* In *24th USENIX Security Symposium (USENIX Security), Washington, DC*, 2015.

[3] Elenbogen, Bruce S. and Seliya, Naeem: *Detecting outsourced student programming assignments.* J. Comput. Sci. Coll., 23(3):50–57, January 2008, ISSN 1937-4771. `http://dl.acm.org/citation.cfm?id=1295109.1295123`.

[4] Jadalla, Ameera and Elnagar, Ashraf: *Pde4java: Plagiarism detection engine for java source code: a clustering approach.* International Journal of Business Intelligence and Data Mining, 3(2):121–135, 2008.

[5] Kothari, Jay, Shevertalov, Maxim, Stehle, Edward, and Mancoridis, Spiros: *A probabilistic approach to source code authorship identification.* In *Information Technology, 2007. ITNG'07. Fourth International Conference on*, pages 243–248. IEEE, 2007.

[6] Lange, Robert Charles and Mancoridis, Spiros: *Using code metric histograms and genetic algorithms to perform author identification for software forensics.* In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 2082–2089. ACM, 2007.

[7] Rosenblum, Nathan, Zhu, Xiaojin, and Miller, Barton P: *Who wrote this code? identifying the authors of program binaries.* In *European Symposium on Research in Computer Security*, pages 172–189. Springer, 2011.

[8] Son, Jeong Woo, Noh, Tae Gil, Song, Hyun Je, and Park, Seong Bae: *An application for plagiarized source code detection based on a parse tree kernel.* Eng. Appl. Artif. Intell., 26(8):1911–1918, September 2013, ISSN 0952-1976. `http://dx.doi.org/10.1016/j.engappai.2013.06.007`.