Unsupervised Learning Based Approach for Plagiarism Detection in Programming Assignments

by

Jitendra Yasaswi Bharadwaj katta, Srikailash G, Anil Chilupuri, Suresh Purini, C V Jawahar

in

Innovations in Software Engineering Conference, ISEC

Report No: IIIT/TR/2017/-1

Centre for Software Engineering Research Lab
International Institute of Information Technology
Hyderabad - 500 032, INDIA
February 2017

Unsupervised Learning Based Approach for Plagiarism Detection in Programming Assignments*

Jitendra Yasaswi
IIIT Hyderabad

Sri Kailash
IIIT Hyderabad

Anil Chilupuri
IIIT Hyderabad

Suresh Purini
IIIT Hyderabad

C. V. Jawahar
IIIT Hyderabad

ABSTRACT

In this work, we propose a novel hybrid approach for automatic plagiarism detection in programming assignments. Most of the well known plagiarism detectors either employ a text-based approach or use features based on the property of the program at a syntactic level. However, both these approaches succumb to code obfuscation which is a huge obstacle for automatic software plagiarism detection. Our proposed method uses static features extracted from the intermediate representation of a program in a compiler infrastructure such as *gcc*. We demonstrate the use of unsupervised learning techniques on the extracted feature representations and show that our system is robust to code obfuscation. We test our method on assignments from introductory programming course. The preliminary results show that our system is better when compared to other popular tools like MOSS. For visualizing the local and global structure of the features, we obtained the low-dimensional representations of our features using a popular technique called t-SNE, a variation of Stochastic Neighbor Embedding, which can preserve neighborhood identity in low-dimensions. Based on this idea of preserving neighborhood identity, we mine interesting information such as the diversity in student solution approaches to a given problem. The presence of well defined clusters in low-dimensional visualizations demonstrate that our features are capable of capturing interesting programming patterns.

Keywords

Source code metrics, Visualization, Code obfuscation, Feature representations, Neighborhood embedding, Unsupervised learning, Plagiarism

*

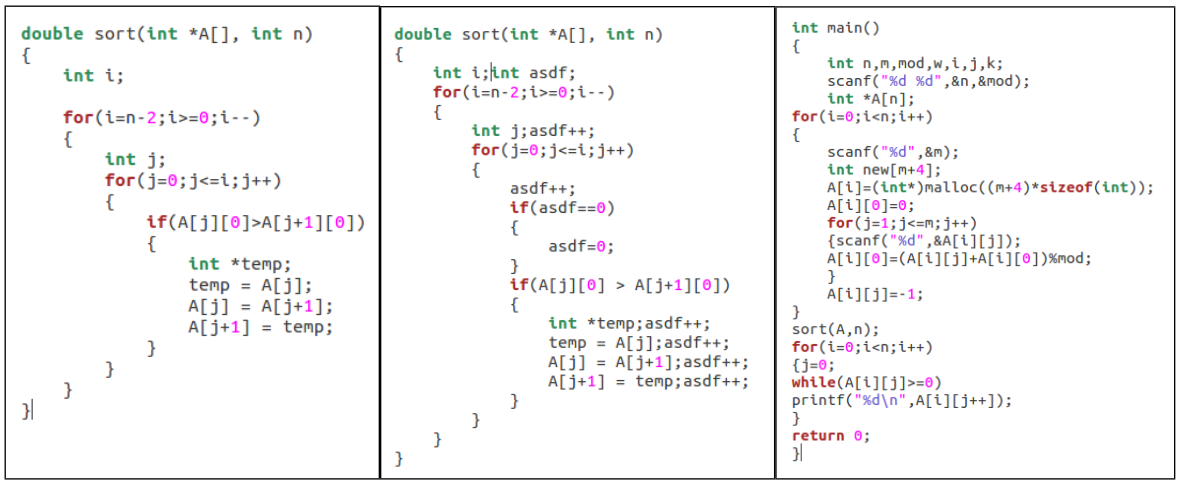
ISEC '17, February 05-07, 2017, Jaipur, India

DOI: http://dx.doi.org/10.1145/3021460.3021473

1. INTRODUCTION

Martins et al. [8] define plagiarism as "the usage of work without crediting its authors". The easy and cheap access to enormous web content has turned plagiarism into a serious problem be it for researchers, publishers or educational institutions. Especially, due to the rapid advancement of technology, handwritten assignments have been replaced by electronic assignments. With the intention of achieving good grades with less or almost no effort, students often try to copy the assignments from their friends. In educational institutions, freshmen who plagiarize in their courses are more likely to continue this malpractice in their later courses. Therefore this malpractice needs to be curbed at its initial stages. The instructor of a course can receive a false feedback about the level of the course and performance of the students. This makes the problem of assignment plagiarism detection an important task. It is hard to manually inspect and (decide whether a submission is genuine or plagiarized) detect similar student submitted solutions in a large class. Though manual inspection is effective, it is laborious and time consuming. One possible way to address this is to seek the help of automated code comparison tools like MOSS [12], JPlag [10] which help in identifying similar submission pairs. Most of the well known automatic comparison tools employ a text-based approach or use the features based on the property of the assignments at a syntactic level to detect plagiarism. However, both these approaches succumb to code obfuscation [9] which is a huge obstacle to automatic software plagiarism detection. Often students use clever techniques to obfuscate the code and evade from being detected. In the context of programming assignments of an introductory computer science course, few examples of code obfuscation are altering a *variable* name, careful conversion of *while* loop into *for* loop and dead code injection etc.

Most of the well known plagiarism detectors like MOSS use a text-based approach or use the features based on the property of the assignments at a syntactic level that use winnowing [12], a local fingerprinting algorithm. MOSS fingerprint selection is not very accurate (selects the fingerprint with minimum value in a window). On top of this fingerprint, a longest common sequence search is performed. Usually when teaching assistants evaluate the student submitted solutions, they consider only the solution pairs as copy cases where similarity score is above some threshold (say 80%). This threshold varies from assignment to assignment depending on the type of problem asked to solve in



```c
double sort(int *A[], int n)
{
    int i;

    for(i=n-2;i>=0;i--)
    {
        int j;
        for(j=0;j<=i;j++)
        {
            if(A[j][0]>A[j+1][0])
            {
                int *temp;
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

(a) Code snippet-1

```c
double sort(int *A[], int n)
{
    int i;int asdf;
    for(i=n-2;i>=0;i--)
    {
        int j;asdf++;
        for(j=0;j<=i;j++)
        {
            asdf++;
            if(asdf==0)
            {
                asdf=0;
            }
            if(A[j][0] > A[j+1][0])
            {
                int *temp;asdf++;
                temp = A[j];asdf++;
                A[j] = A[j+1];asdf++;
                A[j+1] = temp;asdf++;
            }
        }
    }
}
```

(b) Code snippet-2

```c
int main()
{
    int n,m,mod,w,i,j,k;
    scanf("%d %d",&n,&mod);
    int *A[n];
for(i=0;i<n;i++)
{
    scanf("%d",&m);
    int new[m+4];
    A[i]=(int*)malloc((m+4)*sizeof(int));
    A[i][0]=0;
    for(j=1;j<=m;j++)
    {scanf("%d",&A[i][j]);
    A[i][0]=(A[i][j]+A[i][0])%mod;
    }
    A[i][j]=-1;
}
sort(A,n);
for(i=0;i<n;i++)
{j=0;
while(A[i][j]>=0)
printf("%d\n",A[i][j++]);
}
return 0;
}
```

(c) Code snippet-3

Figure 1: Sample code-snippets showing the dead code injection, one of the most commonly adopted technique by students to obfuscate the code. Carefully note the usage of the variable `asdf` in code snippet-2.

the assignment. The approaches proposed in [4] is based on searching similar n-grams or small character sequences (strings) between two source codes. In [11] the authors proposed a representation of source code pairs by using five high level features; namely: lexical features, stylistic feature, comments feature, programmer's text feature, and structure feature. Particularly, for the lexical, comments and programmer's text features, they represent source code as a set of characters n-grams. These features are more oriented to detect aspects that the programmers leave in natural language more than in a particular programming language [11]. In one way or other, each of the above works focuses on the source code file content. However these type of approaches fail against code obfuscation. With minor changes students can successfully evade from getting detected as copy cases. For example consider the code-snippets shown in fig. 1. Both these code-snippets are for sorting. However in fig. 1b, the student who wrote this code intelligently added the dummy variable *adsf*, injected a conditional block and increments its value, which are of no utility. This is one simple example of code obfuscation. Our proposed method successfully detects these type of copy cases. More analysis and performance of our method on this type of plagiarized cases is mentioned in Section 3.4.

In this work we propose a novel hybrid approach to address automatic plagiarism detection. A common approach of the prior works mentioned here is to use code-features to detect plagiarism in programming assignments. In [3], the authors gathered a set of measurable features and trained a neural network on a set of hand-tagged assignment submissions. Instead of hand-tagging and using the labels (supervised learning), we try to cluster (unsupervised learning method) the similar student solutions based on their similarity. However, they make use of certain text-based features like string literals, misspelled comments along with results from other existing plagiarism detectors like MOSS, restricting to twelve features in total. Moreover, it requires considerable human effort to label these assignment submissions and it is difficult for humans to detect partially plagiarized cases. Our idea is to extract features from the code during compilation and make use of them, which can capture the variations observed in the code as belonging to three distinct themes: *structure*, *syntax* (syntax refer to the tokens that occur within basic blocks) and *presentation* as mentioned in [6]. The key contributions of our work are:

- Use of source code metrics (static code-based features) extracted during code compilation as feature representations of the the student solutions to the given programming assignments.
- Unsupervised learning based approach to detect potential plagiarized cases.

In Section 2 we describe our proposed approach, feature extraction and description. In Section 3 we describe the datasets used, the experiments, feature visualization, present our preliminary results and discuss about specific cases where we perform better than MOSS. We mention our future work in Section 4 and conclude our work in Section 5.

2. APPROACH

Given a programming problem to solve from an introductory computer science course and a set of corresponding correct student submitted solutions written in C language. The task is to automatically detect all the plagiarized submission pairs. Our proposed method automatically detects the solution pairs that are most susceptible to be the plagiarized pairs or the cheating cases. Our method accepts as input a set of correct student solutions. Let $\{x_2, x_2, \ldots, x_m\}$ be the student submissions for a given problem, one submission per student. We extract source code metrics from the student solutions, use them as feature representations so that each student solution is mapped to a point in an n-dimensional (here $n = 55$) space. The feature representations from the solutions are then compared pairwise (computing for each pair a total similarity value). We consider student solutions that lie close to each other to be possible plagiarized

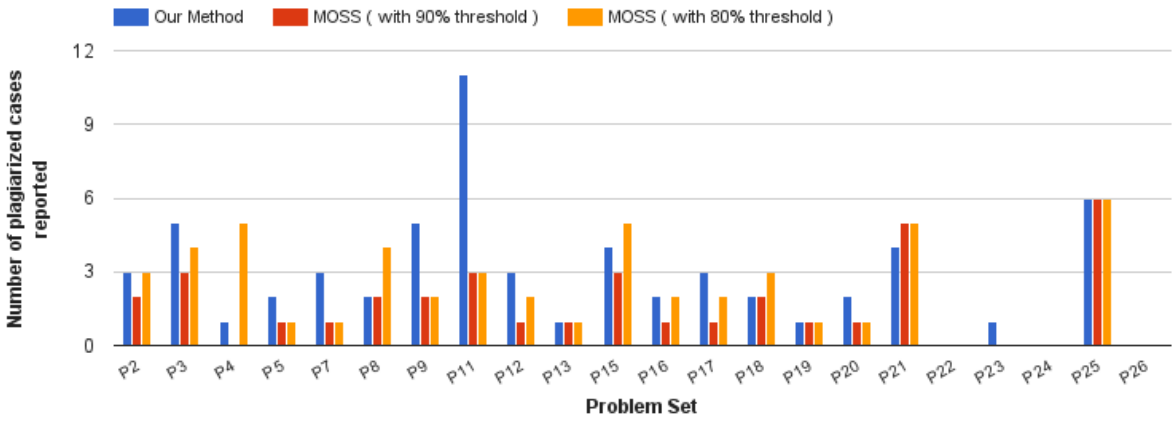


Figure 2: A plot showing the performance of our method when compared with MOSS. The blue colored bars represent the performance of our method, the red and yellow bars represent the MOSS performance when the threshold on score is 90% and 80% respectively.

cases. The closeness or similarity is defined by the Euclidean distance measure between candidate solution pairs which is given by

$$d(a,b) = \sqrt{(a_1-b_1)^2 + (a_2-b_2)^2 + \ldots + (a_n-b_n)^2}, \quad (1)$$

where a, b are two different student solution points in n dimensional Euclidean space. Based on the pairwise Euclidean distance, we cluster together the similar solutions. If the pairwise euclidean distance value for a pair of solutions is less than some threshold (δ), then these pairs are more likely to belong to the same cluster. For acceptable values of δ (say $\delta = 0$), the pairs are either full copy case or partial copy cases. At least there is some similarity in the logic in the submitted solutions. For more details please refer to the discussion section.

2.1 Feature Extraction and Description

Our idea is to design and use features that can alleviate the effect of code obfuscation and be able to detect the plagiarized student solutions. With this intention in mind, we arrived at the idea of using source code metrics as feature representations for programming solutions. In order to avoid getting caught as plagiarized cases, students change the text of the code as that is the simplest thing that can be done with very less effort. However, the internal logic and implementation is many times the same and it can be captured by the intermediate representations after compilation (like edges in control flow graph). Instead of using text based approach, here we use source code metrics as static code-based features that are extracted using MILEPOST GCC [1] feature extraction plugin. MILEPOST framework transforms GCC into a powerful machine learning enabled research infrastructure suitable for adaptive computing. It uses program feature extractor to modify internal optimization decisions. MILEPOST GCC version2.1 can extract sixty five static features in total. These extracted features depends on the type of code optimized selected by the user. Out of these sixty five dimensions features we choose the first fifty five features as our feature vector.

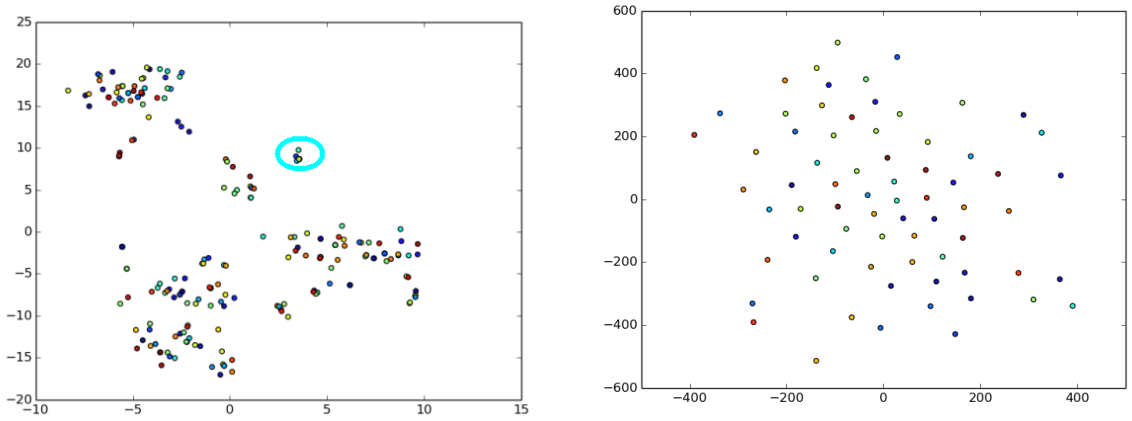
The features in our feature vector can be roughly divided into four subsets depending on the type of program characteristic they capture. The first subset consists of twenty three features which are *basic block* features. They describe a given program based on the number of basic blocks, basic blocks with successors, predecessors etc. The next subset consists of three features that can be termed as *control flow graph* features. These three features describe edges, critical edges and abnormal edges in the control flow graph. The next subset of eighteen features are *method features*. They contribute in capturing information related to methods. The last subset contain features that capture characteristics like occurrence of integer constants, static/local variables etc. For a complete list of features and their description please refer here [2]. Consider the code snippets shown in fig. 1. The *main* method for both the code snippets is shown in fig. 1c. The first feature (ft1) refers to the number of basic blocks in the method which are 15. The number of edges in the control flow graph for both the code snippets is 18, which is captured by our sixteenth feature (ft16). One can observe there are no static/extern variables referred in the main method which assigns the value of our feature 52 (ft52) with a zero. These simple instances mentioned above prove that our features are able to capture variations in the code at a structural and presentation level.

Once we have features representing student solutions, we turn to classical unsupervised machine learning techniques like clustering to complete the task of plagiarism detection. Unlike supervised learning techniques which requires labeled data, unsupervised learning techniques finds a structure in a collection of unlabeled data. We use pairwise euclidean distance between the features as a distance metric to form clusters of similar student solutions.

3. EXPERIMENTS

3.1 Datasets

The dataset we adopted is a collection correct student solutions of assignments (each assignment is a problem set with a single problem) from an introductory C programming course. There are 22 problem sets, each problem set



(a) 2-DIMENSIONAL EMBEDDING

(b) 2-DIMENSIONAL EMBEDDING

Figure 3: A scatter plot showing the low-dimensional embedding obtained by using t-SNE. Observe the clustered points circled in blue in fig. 3a. These are the student solutions that used similar logic to solve the given sorting problem. Observe the spread of the data in both the figures. The students solutions in fig 3b are more diverse.

containing about 70 to 250 student submissions. Totally there are nearly 4,700 submissions. The questions asked in the problem sets range from more specific (as in case of tree-traversal) to diverse. Each problem set is of varying difficulty, with student solutions ranging from 50 to 400 lines of code.

3.2 Experimental Details

As mentioned in section 2.1, we extracted features representations for all solutions from all the problem sets. The pairwise Euclidean distance is calculated using the features. As a preliminary work, we have identified that the solution pairs as copy cases by using a distance threshold $\delta = 0$. With the current threshold we are able to identify exact copy cases as well as some partial copy cases. We use MOSS score as a baseline. The results are shown in fig. 2.

3.3 Feature Visualization

Having proposed source code metrics as feature representations of student solutions, we are interested in answering the question "*What information is captured by the extracted static code-based features and what it is not capturing ?* ". For this purpose, we used t-SNE [7], which can help us in developing and evaluating our feature representations. For visualizing the local and global structure of the feature representations, we obtained the low-dimensional representations (a two-dimensional map in Euclidean space) of our features using t-SNE [7], a variation of Stochastic Neighbor Embedding [5], which can preserve neighborhood identity in low-dimensions. The intuition to use t-SNE came from its inherent ability to preserve neighborhood identity and semantic similarity in low-dimensions. We claim that if our feature representations are good enough, clustering using these representations should give us well defined clusters, each cluster containing all the student solutions that used similar logic to solve a given problem. We arrived at some interesting conclusions like finding out explicit *diversity* in student solution approaches i.e. different ways in which students developed a solution to the given problem. The presence of well defined clusters in low-dimensional visualizations demonstrate that our features are capable of capturing this interesting result. For example, consider the scatter plots shown in fig. 3. The mapped student solutions in the cluster circled in blue color used two variables to achieve swapping. Similarly, they make use of register variables to speed up sorting. While, all other solutions used three variables to achieve swapping. The student solutions corresponding to fig. 3b are more diverse. The spread of the data in fig. 3b along both the axis is more when compared to spread of data in fig. 3a. The spread of the features in the two-dimensional map itself reveals the nature of the solution submitted (the scatter plot looks cluttered for a problem which can be solved in a limited number of ways and wide spread for a problem which can be solved in multiple ways).

3.4 Preliminary results and Discussion

In most of the cases, our results are consistent with scores from MOSS. The potential plagiarized solution pairs detected by our method falls in top-five cases detected by MOSS. There are solution pairs where MOSS does not show any similarity and the pairwise distance is comparatively very low. However, our proposed method does a better job in most cases. From fig. 2, we can observe the results of our method compared with MOSS. When comparing our method with MOSS, we have to decide two thresholds namely: a threshold on pairwise Euclidean distance and a threshold on the obtained MOSS scores. Depending on the domain knowledge (here the type of problem asked to solve in the problem set) these thresholds vary. The threshold on pairwise Euclidean distance selected we used is zero ($\delta = 0$). The thresholds on MOSS scores is 90% and 80% respectively. Depending on the distance threshold (δ), we can report more confident and refined accurate results. We an-

alyze our results in the cases presented below:

Case1: *Absence of plagiarized cases*
From fig. 2, we can observe that there are no reported copy cases for problem sets P22, P24 and P26. Similarly, for these problem sets, the similarity scores of MOSS are also very low (less than 50%).

Case2: *Interchanging if-else code*
If the code contains *if-else* blocks and if the conditions are interchanged then MOSS does not give us any acceptable similarity score as a plagiarized case. However, our feature vector does not differ in this case.

Case-3: *Type define the frequently called functions*
In one of code if the frequently called functions like *printf* and *scanf* are type defined, then MOSS shows only 50-60% but it should be a 100% copy case. In this scenario, our method works perfectly well, resulting in a pairwise distance value of zero.

Case-4: *Presence of dead code*
If dead code is added to one of the codes the MOSS score is very low. In such cases, compiler optimization can be used to extract the features which can successfully eliminate the effect of dead code. The example in figure 1b shows a code snippet containing dead code.

Case-5: *Interchange the position of functions*
If function code is interchanged MOSS shows 60-70% for copy pairs for the exact match of the code. However, our measured pairwise distance is zero which makes it easy to detect the copy cases.

All the above five cases provide enough proof about the robustness of our features and its ability to detect plagiarized cases even in obfuscated solution pairs. This demonstrates the superiority of our method when compared to MOSS.

4. FUTURE WORK

Our proposed work in this paper is in its initial stages. We identified promising directions in which this work can be extended. In future, we would like to address the following:

- Identify and use additional dynamic features that could boost the performance of our method.
- Proposing a method to decide a good distance threshold (δ), which enables to detect partial plagiarized solution pairs confidently.

5. CONCLUSION

Static program features extracted from the intermediate representations during the various phases of compilation process are used successfully to address the compiler phase optimization problem. In this work, we explored the possibility of using those features for plagiarism detection and, mining programming patterns and the associated visualizations. Our initial experiments suggests that the approach is very promising and can be used in many different ways.

6. REFERENCES

[1] http://ctuning.org/wiki/index.php/CTools:MilepostGCC.

[2] http://ctuning.org/wiki/index.php/CTools:MilepostGCC:StaticFeatures:MILEPOST_V2.1.

[3] S. Engels, V. Lakshmanan, and M. Craig. Plagiarism detection using feature-based neural networks. *ACM SIGCSE Bulletin*, 39(1):34–38, 2007.

[4] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno. Towards the detection of cross-language source code reuse. In *International Conference on Application of Natural Language to Information Systems*, pages 250–253. Springer, 2011.

[5] G. E. Hinton and S. T. Roweis. Stochastic neighbor embedding. In *Advances in neural information processing systems*, pages 833–840, 2002.

[6] A. Luxton-Reilly, P. Denny, D. Kirk, E. Tempero, and S.-Y. Yu. On the differences between correct student solutions. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 177–182. ACM, 2013.

[7] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[8] V. T. Martins, D. Fonte, P. R. Henriques, and D. da Cruz. Plagiarism detection: A tool survey and comparison. In *OASIcs-OpenAccess Series in Informatics*, volume 38. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[9] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu. Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. 2016.

[10] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016, 2002.

[11] A. Ramírez-de-la Cruz, G. Ramírez-de-la Rosa, C. Sánchez-Sánchez, H. Jiménez-Salazar, C. Rodríguez-Lucatero, and W. Luna-Ramírez. High level features for detecting source code plagiarism across programming languages.

[12] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.