# Improving Plagiarism Detection in Coding Assignments by Dynamic Removal of Common Ground

**Christian Domin**
University of Hannover
Hannover, Germany
christian.domin@hci.uni-hannover.de

**Henning Pohl**
University of Hannover
Hannover, Germany
henning.pohl@hci.uni-hannover.de

**Markus Krause**
ICSI, UCBerkeley
Berkeley, CA, USA
markus@icsi.berkeley.edu


*CHI'16* Extended Abstracts, May 07-12, 2016, San Jose, CA, USA
ACM 978-1-4503-4082-3/16/05.
http://dx.doi.org/10.1145/2851581.2892512

## Abstract
Plagiarism in online learning environments has a detrimental effect on the trust of online courses and their viability. Automatic plagiarism detection systems do exist yet the specific situation in online courses restricts their use. To allow for easy automated grading, online assignments usually are less open and instead require students to fill in small gaps. Therefore solutions tend to be very similar, yet are then not necessarily plagiarized. In this paper we propose a new approach to detect code re-use that increases the prediction accuracy by dynamically removing parts in assignments which are part of almost every assignment—the so called common ground. Our approach shows significantly better F-measure and Cohen's $\kappa$ results than other state of the art algorithms such as Moss or JPlag. The proposed method is also language agnostic to the point that training and test data sets can be taken from different programming languages.

## Author Keywords
Plagiarism; computer science education; massive open online courses

## ACM Classification Keywords
K.3.2 [Computers and Education]: Computer science education.

## Introduction

While MOOCs can increase access to education, there is concern that the openness of these course result in fraud. One often cited fraud is plagiarism as pointed out by Cooper and Sahami [2]. With much larger courses compared to offline classes, manually checking students' work is not feasible. While some providers try to increase the level of control by having students visit dedicated test centers, this also increases costs for students. Plagiarism detection is a possible solution to this challenge.

In programming assignments, it's common to provide some template code to students, which will then be contained in almost all submissions. This part of the code forms the so-called common ground. If it is not considered, actual plagiarism can hide between valid code reuse. However, this is not the only value of the common ground. Simple tasks and code conventions (such as the name of a control variable in loops) also lead to common ground. As described by Mann and Frew [7] there are several more reasons why student programming assignments might by similar. So one important requirement of this tools is the reliable removal of the common ground in all submissions.

Common plagiarism tools (such as [10, 12, 1]) remove common ground with fixed probability. So a code fragment will not be considered after it occurs x-times in all submissions. In general, the students who plagiarize their submission try to mask this by making some changes.

As described by Faidhi and Robinson [4], weak students often only make small changes like renaming some variables, or exchanging some lines of code without modifying the semantics at all. This includes changes on the common ground as well. Figure 1 shows an example of such a template file.

Additional these tools provide often the possibility to submit a common ground file. This enables the tool to ignore specific code fragments which were given with the assignment. This approach is a good option, but sometimes such a common ground file is not available. Code conventions or usual solutions can not be covered.

We introduce a method to dynamically remove common ground in code submissions and compare this approach to state of the art plagiarism detection software. Furthermore we answer two more specific research questions:

**R1** Can dynamic common ground removal improve prediction quality?

**R2** Can cross training improve results when the training data for one language is sparse?

## Using Common-Ground Removal to Improve Code Re-Use Detection

Here we describe a metric which is used to specify the similarity between two submissions. We also show how a random forest tree can be trained to detect code re-use, based on that metric.

We use pairwise comparison of submissions. So a set of $N$ submissions (denoted as $\mathbb{S}$) leads to $\frac{N \cdot (N-1)}{2}$ pairs to compare. Note that this detects code re-use, but does not distinguish which submission copied from the other.

In a first step, we preprocess all submissions to transform them into an n-gram representation—sets of small code fragments with a length of $n$. The advantage of this method is that permutations in the code lead to identical values in the similarity metric. So it is not possible to mask plagiarism by exchanging some lines of code. We store n-grams in a multiset to preserve how often they occur in a submission.

**File 1:** Submission A

```
/** Read in query statements */
public static void readQueries () {
 // read in one integer
 int m = sc.nextInt();
 // loop to read pairs of [...]
 for (int i=0;i<m;i++) {
   int start = sc.nextInt();
   int end = sc.nextInt();
   if (tree.isPathSafe(start,end)==true)
    System.out.println("YES");
   else
    System.out.println("NO");
 }
}
```

**File 2:** Submission B

```
/** Read in query statements */
public static void readQ () {
 // read in one integer
 int j = sc.nextInt();
 // loop to read pairs of [...]
 for (int i=0;i<j;i++) {
   int start = sc.nextInt();
   int end = sc.nextInt();
   if (tree.isPathOk(start,end)!=true)
    System.out.println("NO");
   else
    System.out.println("YES");
 }
}
```

**File 3:** Template File

```
/** Read in query statements */
public static void readQueries () {
 // read in one integer
 int m = sc.nextInt();
 // loop to read pairs of [...]
 for (int i=0;i<m;i++) {
   sc.nextInt(); // modify this line
   sc.nextInt(); // modify this line
  }
}
```

**Figure 1:** Plagiarized submissions with marked code lines out of the template file. Submission A uses more code of the template file. In contrast submission B does a lot of changes at the template code lines. Removing these lines in submission A would reduce the similarity and lead to information loss. These submissions are out of the JAVA corpus described in [9].

Our metric should express which parts in each submission are individual and which are reused. Thus, for each n-gram $\eta$ we define the probability $p$ as how likely it is the same n-gram appears in other submissions (irrespective of how often it appears in them) as:

$$p(\eta) := \frac{|\{s \mid s \in \mathbb{S} \wedge \eta \in s\}|}{|\mathbb{S}|}.$$

We can now remove all n-grams that are equal or more likely than some threshold probability $p$. Where $s$ denotes a submission, we use $s'$ to represent a submission filtered in this way.

In comparison to other plagiarism detection tools like *Moss* [12], this algorithm doesn't use a fixed probability at which a code fragment won't be considered anymore. The similarity metric of the algorithm is parameterized with the probability threshold $p$ and the n-gram size $n$.

The similarity of two submissions $s_a$ and $s_b$ is given as:

$$sim(s_a, s_b, n, p) := \begin{cases} \dfrac{|s'_a \cap s'_b|}{\min(|s_a|, |s_b|)} & \text{if } s'_a = \emptyset \vee s'_b = \emptyset \\ 0 & \text{else} \end{cases},$$

where $n$ describes the n-gram length. The intersection contains each n-gram as often as the minimum occurrence in one of the sets. The cardinality of the intersection depends on the length of the submissions. Hence, a high value of the cardinality not always leads to a high similarity in the sense that long submissions in general tend to have high frequency values here. This problem can be avoided by using a factor based on the maximum possible cardinality. The extreme case is when one submission completely contains another. In that case the cardinality equals the frequency of all n-grams in the shorter submission.

Ideal parameter choice of $n$ and $p$ varies between different assignments. They are based on how much code is given in the assignment or how much different solutions are common for that task. We chose to take this variation into account, and train a random forest classifier for this task. As feature vector this classifier uses:

$$\begin{pmatrix} sim(s_a, s_b, n_0, p_0) \\ \vdots \\ sim(s_a, s_b, n_m, p_m) \end{pmatrix}.$$

By computing similarity for different parameter values, and including all those measures in the final feature vector, we enable the classifier to take into account this variation.

Our final vector includes n-grams lengths of 3, 10, and 20 (see Figure 2 for an example comparison of how n-gram size impacts similarity). We furthermore include ten different thresholds for $p$: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0. Finally, we also include the special case of not removing any common ground.

By having $p$ vary, we can remove the common ground dynamically. With decreasing probability $p$ the amount of code that will be removed increases. At some value for $p$ we remove plagiarized code parts in the submissions. The similarity between plagiarized submissions is now low as well. It is not possible anymore to separate reused code and original submissions. We do not consider the value $p = 0.0$, because when removing all code, no separation between submissions is possible (see Figure 3).

## Measurements

To compare our results we use F-measures. As F-measure is sensitive to unequal class distributions we also report Cohen's Kappa. In both cases we report standard deviations where possible. To investigate differences we use Pearson's chi-squared test.

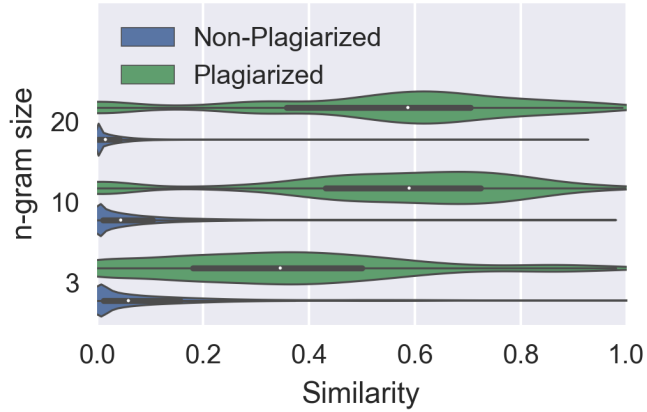


**Figure 2:** This plot shows the similarity distribution for three n-gram sizes considering all thresholds for $p$ (using the JAVA training corpus [5]). We find a good separation of the plagiarized and non-plagiarized submissions for n-gram sizes 10 and 20. The pattern changes for different corpora.

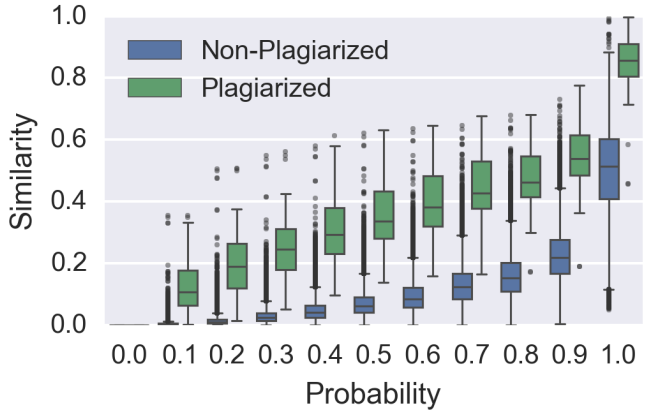


**Figure 3:** Plotting similarity (also using submissions from [5]) for different thresholds of $p$ (with a fixed $n = 3$) shows the benefit of dynamic common ground removal.

## Results

To compare our approach we use the training corpus provided by the *SOCO 2014* challenge [5]. The corpus contains submissions in Java ($n = 259$, 26 incidents of code reuse) and C ($n = 79$, 84 incidents of code reuse).

We report on three different results of our experiment the difference between our approach and a set of baseline methods 2) the differences between our dynamic removal approach and our static approach and 3) the effect of cross training the classifier with samples from one language and predicting outcomes of samples written the other language.

*Baseline comparison*

We compare our dynamic approach to three baseline approaches Moss, JPlag (results taken from [5]), and the best contender of the SOCO challenge as reported in [5]. We obtained the results for Moss from its online tool [1]. Table 1 shows F-measure means and standard deviations of all tested approaches. The results indicate that our dynamic approach has a better performance for the given dataset than any other approach. The Kappa results in table 2 show that for the given data set our approach has a good balance of true positive and true negative results. Finally, Table 3 shows that the differences between Moss and our approaches are significant.

*Dynamic vs. Static Removal*

As Table 2 indicates our dynamic removal approach shows better results than the static approach for all data sets. The difference between both approaches are significant as shown in Table 3.

[1] https://theory.stanford.edu/ aiken/moss/

| | C (SD) | JAVA (SD) |
|---|---|---|
| JPlag | 0.30 (–) | 0.55 (–) |
| Moss | 0.37 (0.05) | 0.55 (0.05) |
| Best [5] | 0.42 (0.03) | 0.66 (0.04) |
| Static | 0.38 (0.03) | 0.75 (0.06) |
| Static (Cross) | 0.46 (0.04) | 0.70 (0.09) |
| Dynamic | 0.47 (0.01) | 0.80 (0.01) |
| Dynamic (Cross) | 0.54 (0.01) | 0.72 (0.01) |

**Table 1:** Average F-measure results and standard deviations of 20 test runs of our approach on two assignment sets (JAVA and C). JPlag results where taken from [5] without SD. The Best row shows the results of the best contender of the SOCO challenge. Please note that the best contender for C is different from the one for JAVA.

| | C (SD) | JAVA (SD) |
|---|---|---|
| Moss | 0.37 (0.03) | 0.53 (0.05) |
| Static | 0.38 (0.03) | 0.73 (0.06) |
| Static (Cross) | 0.43 (0.04) | 0.68 (0.08) |
| Dynamic | 0.47 (0.01) | 0.79 (0.01) |
| Dynamic (Cross) | 0.52 (0.01) | 0.71 (0.01) |

**Table 2:** Average Cohen's $\kappa$ results and standard deviations of 20 test runs of our approach on two sample sets (JAVA and C). The rows marked with (Cross) indicate cross training results.

| | Java Samples | | C Samples | |
|---|---|---|---|---|
| | $\chi^2$ | p | $\chi^2$ | p |
| S vs. D | 13K (11K) | $<$0.001 | 245 (317) | $<$0.001 |
| S vs. M | 13K (12K) | $<$0.001 | 225 (182) | $<$0.001 |
| D vs. M | 14K (8K) | $<$0.001 | 129 (285) | $<$0.001 |

**Table 3:** Results of Pearson's $\chi^2$ tests comparing the three different approaches (S)tatic, (D)ynamic, (M)oss. Numbers in round brackets are for the cross training results. All p-values are below the 0.001 $\alpha$-level.

*Cross Training*
As Table 2 indicates cross training does not deteriorate the results of our approach. In fact cross training enhance some results. The classifier trained on Java samples to predict code reuse in C samples shows better results than the classifier trained on Java samples. The reason for this is most likely the fact that the Java samples have more code reuse samples (84) than the C samples (24).

## Conclusion

We introduced an algorithm which improves the common ground removal in student programming assignments. The algorithm is language independent and does not require structural information, which would need to be provided for every target language (this is an advantage over structure-metric approaches, such as [11]). It uses a random forest tree to classify each pair of submissions as either plagiarism and original submission. With dynamic common ground removal there is no need for educators to submit a template to mark common code fragments.

Additionally, dynamic common ground removal covers code fragments which might be similar for other reasons (following some code conventions, or similar solutions for some easy task). This also avoids the removal of hints towards plagiarism which are supported by the common ground. Students might, e.g., change the common ground to mask plagiarism. Hence, removing it in all other submissions would hide this similarity.

For the given data set, our approach is better than other state of the art algorithms. F-measure and Cohen's Kappa results obtained indicate that our approach is significantly better than Moss even without dynamic common ground removal. Compared to JPlag our algorithm shows 25%-80% F-measure increase. Compared to the best results reported by [5] the increase is between 10% and 28% when used with cross training. We can thus answer R1 positively as dynamic common ground removal shows significantly better Kappa results than the static approach. Finally we can conclude that for the given data set cross training can help obtaining significantly better results when the training data is sparse (R2).

Our system computes pairwise similarities between submissions. This already is a valuable aid for educators and reduces the set of pairs they need to manually validate. Existing clustering approaches for plagiarism detection (such as [8]) could be run on top of our system to further help support this task.

This overall shows, that our dynamic approach to common ground removal can boost performance in plagiarism detection. Such improved plagiarism detection can increase trust in assignments of MOOCs or other large classes. Reliability of plagiarism detection is a vital component when giving credit for online classes, ensuring the quality of the work being done (relying on an honor code is not sufficient [3]). Such quality improvement is especially important for students without access to traditional education offerings, and has the potential to increase their retention rate (a problem in MOOCs [6]).

## References


[1] Ahtiainen, A., Surakka, S., and Rahikainen, M. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling* (2006), 141–142.

[2] Cooper, S., and Sahami, M. Reflections on Stanford's MOOCs. *Commun. ACM 56*, 2 (Feb. 2013), 28–30.

[3] Corrigan-Gibbs, H., Gupta, N., Northcutt, C., Cutrell, E., and Thies, W. Measuring and Maximizing the Effectiveness of Honor Codes in Online Courses. In *Proceedings of the Second ACM Conference on Learning @ Scale* (2015), 223–228.

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

[5] Flores, E., Rosso, P., Moreno, L., and Villatoro-Tello, E. PAN@FIRE: Overview of SOCO Track on the Detection of SOurce COde Re-use. In *Sixth Forum for Information Retrieval Evaluation* (2014).

[6] Krause, M., Mogalle, M., Pohl, H., and Williams, J. J. A playful game changer: Fostering student retention in online education with social gamification. In *Proceedings of the second ACM conference on Learning @ scale - L@S '15* (2015).

[7] Mann, S., and Frew, Z. Similarity and Originality in Code: Plagiarism and Normal Variation in Student Assignments. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (2006), 143–150.

[8] Moussiades, L., and Vakali, A. PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets. *The Computer Journal 48*, 6 (2005), 651–661.

[9] Poon, J. Y., Sugiyama, K., Tan, Y. F., and Kan, M.-Y. Instructor-Centric Source Code Plagiarism Detection and Plagiarism Corpus. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (2012), 122–127.

[10] Prechelt, L., Malpohl, G., and Philippsen, M. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal COmputer Science 8*, 11 (2002), 1016–1038.

[11] Rosales, F., García, A., Rodriguez, S., Pedraza, J. L., Méndez, R., and Nieto, M. M. Detection of Plagiarism in Programming Assignments. *IEEE Transactions on Education 51*, 2 (May 2008), 174–183.

[12] Schleimer, S., Wilkerson, D. S., and Aiken, A. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), 76–85.