TMOSS: Using Intermediate Assignment Work to Understand Excessive Collaboration in Large Classes

Lisa Yan
Stanford University
yanlisa@stanford.edu

Nick McKeown
Stanford University
nickm@stanford.edu

Mehran Sahami
Stanford University
sahami@cs.stanford.edu

Chris Piech
Stanford University
piech@cs.stanford.edu

ABSTRACT

As computer science classes grow, instructor workload also increases: teachers must simultaneously teach material, provide assignment feedback, and monitor student progress. At scale, it is hard to know which students need extra help, and as a result some students can resort to excessive collaboration—using online resources or peer code—to complete their work. In this paper, we present TMOSS, a tool that analyzes the intermediate steps a student takes to complete a programming assignment. We find that for three separate course offerings, TMOSS is almost twice as effective as traditional software similarity detectors in identifying the number of students who exhibit excessive collaboration. We also find that such students spend significantly less time on their assignment, use fewer class tutoring resources, and perform worse on exams than their peers. Finally, we provide a theory of the parametric distribution of typical student assignment similarity, which allows for probabilistic interpretation.

CCS CONCEPTS

• **Social and professional topics → Computer science education**; **CS1**; **Student assessment**;

KEYWORDS

Programming courses; plagiarism detection; student performance; teaching at scale; undergraduate courses

ACM Reference Format:
Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. 2018. TMOSS: Using Intermediate Assignment Work to Understand Excessive Collaboration in Large Classes. In *Proceedings of The 49th ACM Technical Symposium on Computer Science Education, Baltimore, MD, USA, February 21–24, 2018 (SIGCSE '18),* 6 pages.
https://doi.org/10.1145/3159450.3159490

1 INTRODUCTION

In the past decade, an increasing number of educators have begun digitizing and transforming the learning experience in order to meet the ever-rising demand for education. This phenomenon is especially apparent in CS education at the undergraduate level, where the nature of the learning material enables efficient use of computing resources. Not only can the content delivery be delivered via online lectures, but also an entire homework assignment can be delivered, submitted, and graded with the aid of autograders.

Due to the size of the CS 1 class at many universities today, it is intractable to monitor every student as they progress through an assignment, even with the use of undergraduate or graduate teaching assistants [5, 19]. Inevitably a large component of the course relies on students performance on assignments, which often must be completed outside of a closed lab setting. Instead of the teacher quickly identifying a student who is struggling, it is often the case that such a student must seek out instructor help through scheduled office hours. At best, teachers can look only at the final homework submission for each student and evaluate whether the student needs additional help based on their final product.

By asking students to take on the responsibility of seeking help, a pressing issue arises. Sometimes, the students who struggle the most will not access the provided learning environments and will instead find their own resources; seeking shortcuts to learning, they can occasionally resort to unpermitted outside resources. This is especially prevalent in large online courses [25].

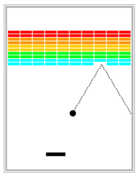
Two questions arise from the current state of large CS 1 classes: Given that there is a risk of *excessive collaboration* on assignments, where students overly rely on outside resources like peer code or online solutions, how can we identify students who exhibit such behavior? Furthermore, how does excessive collaboration correlate with student assignment work patterns and overall course performance? If we can address these two questions, we can learn more about our students, detect students exhibiting excessive collaboration, and ensure a healthy learning environment.

In this paper we tackle both of these problems by introducing Temporal Measure of Software Similarity (TMOSS), based on the well-known Measure of Software Similarity (MOSS) system [20]. TMOSS is a tool that builds on traditional software *similarity score* measures like MOSS. Instead of looking only at a student's final submission, TMOSS computes similarity on intermediate assignment work. These summaries are then verified by a human to hypothesize which students may have exhibited excessive collaboration. Through this procedure, we demonstrate the effectiveness of the TMOSS algorithm and paint a picture of how excessive collaboration impacts student performance. We further provide a theoretical foundation for interpreting TMOSS and MOSS scores and show that the distribution of similarity scores that do not exhibit excessive collaboration can be modeled with a parametric Gumbel distribution. Such a distribution enables formal analysis and also indicates that the likelihood of false positives in our tool is very low.

Our paper proceeds as follows: After summarizing related work in Section 2, in Section 3 we describe our learning environment and

SIGCSE '18, February 21–24, 2018, Baltimore, MD, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159490

Table 1: Statistics per Breakout repository (1420 students).



	Mean	SE
Start day	-5.22	2.73
# Snapshots	253	199
Hours on task	9.77	4.93
TA hours	4.18	9.09

the process of collecting snapshots of student progress through a programming assignment. Section 4 introduces the TMOSS tool, which uses intermediate student work to compute high similarity scores; these are then used to hypothesize excessive collaboration. Section 5 gives a theoretical framework for the interpretation of similarity scores. Finally, in Section 6 we report our results using TMOSS for three course offerings, and we demonstrate that students that exhibit excessive collaboration and those that do not are meaningfully different in how they interact with the class material—including time on task, use of office hours, and exam performance.

2 RELATED WORK

Many software similarity systems have been developed over the past two decades to detect for software plagiarism [7, 18, 20, 23, 24]. Among these systems, MOSS [20] is commonly used in academic settings to detect student plagiarism in course assignments. People have used biometrics measures like keystroke logging to identify plagiarism [10], which is used in Coursera [11].

Online learning and teaching computer science courses at scale have also prompted new research on modeling and understanding student learning from programming assignment solutions. Spohrer et al. [22] observe and collect student progress based on observation of student programming bugs, whereas Piech et al. [16] use assignment progress repositories to analyze intermediate student work. Social science research in the past has attempted to explain motivations for student plagiarism, citing that plagiarism is most common when there are small penalties and high rewards [3, 13].

Recent work has combined these two areas of research and analyzed how excessive collaboration affects student performance. Pierce et al. [17] developed a tool that correlated plagiarism with negative performance over other assignments in the course, and there has also been anecdotal evidence that over-reliance on outside help can incite negative student experiences in a CS 1 course [15]. Schneider et al. [21] implemented a plagiarism detection tool that analyzes logs of student interaction with course software. To the best of our knowledge, our work is the first to use intermediate student work for identifying cases of excessive collaboration and the first to model the probability of false positives in similarity software detection.

3 DATA

In this study we focus on the first large programming assignment in an undergraduate level CS 1. Breakout, a classic Atari game [14], is a programming assignment used widely in many institutions like Stanford, Cornell, Johns Hopkins, and CodeHS [4, 6, 8, 19]. In the course studied, Breakout is the first large, intensive creative project assignment, and enthusiastic students often extend their work beyond the minimum requirements. At the same time, the assignment unfortunately also creates the first opportunity for excessive collaboration.

In the CS 1 course studied, students complete Java programming assignments individually using the Eclipse Integrated Development Environment (IDE). The IDE has been modified to automatically log snapshots of student code in a local git repository every time a student compiles their assignment project to check program output, which occurs whenever the student tries to run their program. This leads to a series of snapshots with temporal granularity on the order of minutes of programming time. Each snapshot in a student code repository can be indexed by a *snapshot index*, corresponding to a timestamped copy of student code at compile time. When students submit their final code, they also submit the code snapshot repository. A repository can therefore be analyzed for functional information on the student's code development path [16] or for temporal information to quantify student work habits.

We identify broad metrics of student work habits: number of hours worked and which day the student started coding. *Similarity scores*—statistics that will be used for measuring collaboration—are then calculated by analyzing the repository in depth (described more in Section 4). Due to the graphics-based, open-ended nature of the Breakout assignment, we unfortunately did not have unit tests or a functioning autograder and did not have metrics on functionality per snapshot.

We use the Breakout assignment submissions from the CS 1 offerings in Fall 2012 (416 students), Fall 2013 (476 students), and Fall 2014 (528 students). All students from this dataset were taught by the same instructor and had nine days to work on Breakout (with an extra two late days to submit with potential grade penalty). The Breakout assignment is held on week four of a 10-week course. While students work on the assignment individually until the deadline, they can also attend walk-in office hours held Sunday to Thursday evenings to clarify concepts or discuss debugging tips with undergraduate teaching assistants (TAs). In our analysis, we leverage attendance logs of these TA hours—students record their check-in time and their assignment or course issue, and TAs record the students' check-out time and issue resolution. Table 1 provides a summary of how these three sets of students worked through the Breakout assignment. On average, students started the assignment 5.22 (SE=2.73) days before the deadline and spent a mean of 9.77 (SE=4.93) hours on task. TA office hours attended were calculated over the entire course. Hours on task were determined by grouping snapshot times that were within half an hour of each other.

4 METHOD

In this section, we give an overview of the typical process of flagging final student submissions for excessive collaboration with other students or online solutions. We then present TMOSS, a tractable method for identifying excessive collaboration over intermediate versions of student code.

4.1 Traditional software similarity detection

Traditional software similarity detectors compute similarity scores over a student's submitted code; these tools compare the student code to peer submissions and online solutions on various metrics,

ALGORITHM 1: Computing top matches in TMOSS.

```
Input: $N$ students (students)
       $n$ online solutions (online)
Output: $N$ top matches (top_matches). Each student has a
        tuple of (highest similarity score, match code).
top_matches = [];
for $i$ in $1 \ldots N$ do
    compare_set =
        getFinalSubmissions(students - students[i]) + online;
    snapshots = getRepository(students[i]);
    student_matches = [];
    for $j$ in $1 \ldots M$ do
        results = compute_similarity(
            snapshots[j], compare_set);
        student_matches.append(
            $\text{argmax}_{k=1 \ldots N+n-1}$ results[k].getScore());
    end
    top_matches.append(
        $\text{argmax}_{j=1 \ldots M}$ student_matches[j].getScore());
end
```

and flag student submissions that deviate from standard statistics [7, 17, 18, 20, 21, 23, 24]. Among these, Measure of Software Similarity (MOSS) is highly regarded as the standard for detecting software plagiarism in the classroom [1, 20]. The software acts as a filtering tool prior to human decision; submissions with high similarity scores are checked manually by course staff, who identify excessive collaboration as plagiarism on a case-by-case basis. These pairwise detectors all scale quadratically with the size of the class, as comparing N student submissions to N peers and $n < N$ known online solutions requires a runtime of $O(N^2)$.

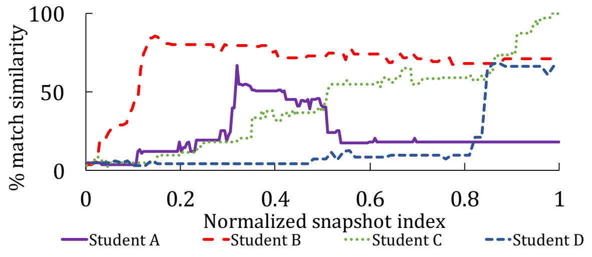
Running these tools on student final submissions can only identify a subset of the students exhibiting excessive collaboration. For example, suppose a student pastes in an online solution momentarily to check the desired output, then removes the online solution and submits his or her own work. While this is a plagiarism case at many academic institutions, existing software similarity detectors cannot flag this student from his or her final submission work.

4.2 TMOSS: Temporal Measure of Software Similarity

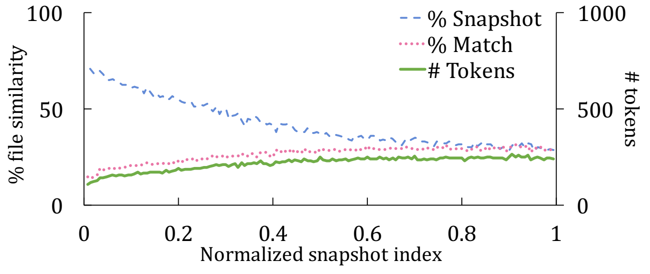
TMOSS is a tool designed to extend traditional software similarity detectors by identifying excessive collaboration on any intermediate snapshot of a student's code repository, not just final code submission. While TMOSS (as the name implies) currently implements MOSS-based similarity scores [20], the tool can be easily adapted to different similarity detector backends.[1]

An outline of TMOSS's operation is shown in Algorithm 1. TMOSS produces N top matches—one per student—corresponding to the final peer or online code that returned the highest similarity score over all M snapshots per student repository. The set of peer submissions are from the same course quarter, whereas online

[1]TMOSS is available as an open-source project at https://github.com/yanlisa/tmoss.



(a)



(b)

Figure 1: (a) Examples of students in the HEC group; (b) 95th percentiles of different MOSS similarity scores over time.

solutions are those that the teaching staff found via github or coding blogs. The `compute_similarity()` function is our traditional similarity detector backend, which computes similarity scores of a student snapshot to each of the final and online code files. Attempting to run existing pairwise similarity detection algorithms on all M snapshots for each of N students would require $O(N^2M^2)$ runtime, which is impractical. Instead, we avoid pairwise comparison by comparing a snapshot to N final peer submissions and $n < N$ online solutions, thus reducing the runtime per snapshot to $O(N)$ and the overall runtime to $O(N^2M)$. We note that comparing snapshots to final code is often preferable; similarity score algorithms that we surveyed scale much better with larger code files, and comparing two intermediate code files often produces a very low similarity score. Finally, we use human detection as the final step to determine which of these N top matches exhibits excessive collaboration. All students who fall into this category are put into the *hypothesized excessive collaboration* (HEC) group.

Figure 1a shows four students from our dataset who fall into the HEC group, using MOSS as the detector backend for TMOSS. The percent (%) similarity of a match's code to the intermediate code snapshot is plotted over time, here computed as the normalized snapshot index (from their first compiled code at 0.0 to their final submission at 1.0). While all of the students graphed exhibit excessive collaboration, Student A would go undetected by a typical run of MOSS on final submissions.

MOSS similarity scores. While any pairwise software similarity detector that fits the function signature of `compute_similarity()` can be used in TMOSS, we describe MOSS, which is used in our implementation [1, 20]. MOSS returns a triplet of similarity scores for each pair of (snapshot, match), where match refers to a final or online code file: (1) number of tokens—MOSS's internal, tokenized representation of code—shared between the two programs, (2) % snapshot similarity, and (3) % match similarity. The latter two scores are computed as the percentage of shared tokens in the tokenized MOSS representation of the student snapshot code (% snapshot similarity) and the matched peer or online code (% match similarity).

In our implementation of TMOSS, we only consider the first of these metrics—the number of shared tokens—to pick the top match for each student. We ignore the % snapshot similarity score since it varies inversely with the length of the student snapshot, and therefore it will be high when the student starts, and will decrease as the student progresses (Figure 1b). By contrast, the other two metrics do not exhibit this behavior. We further compare the effectiveness of number of tokens and % match similarity as similarity scores in Section 6.

5 THEORY

What is the chance that a student, who did not cheat, is reported as having plagiarized? In this section we provide a theoretical framework for calculating the probability of a false-positive maximum similarity score. The theory presented applies to all similarity measures for plagiarism detection, including MOSS and TMOSS.

Consider a student S who worked independently. When we compute a similarity score X_i between student S and another student i, there is an non-zero probability that the score will be accidentally large. While we suppose that this probability is exceedingly small, the likelihood of a false-positive report for student S increases when we compute a similarity score between them and *every other* student in the history of the course. The score Y that is analyzed for student S is the highest similarity score between the student and all other submissions: $Y = \max_i X_i$, for all other students i. The potential for a large max similarity score arising by chance—in the absence of collaboration—is concerning and worth exploring in detail.

We assume that the probability distribution of X_i is unknown, but that it is exponentially-tailed.[2] We also assume that, as the student did not collaborate with any of their peers, the values X_i should be mutually independent. Finally, it is reasonable to suppose that the X_i scores have the same (though unknown) distribution. The Fisher-Tippett-Gnedenko Theorem, a more obscure cousin of the Central Limit Theorem, tells us that the **max** of exponentially-tailed independent identically distributed (IID) variables can only converge to a *Gumbel* distribution [9]. Since we assumed that Y is the max of exponentially-tailed IID random variables, Y should have a Gumbel distribution; as such,

$$\Pr(Y \geq k) = 1 - e^{-e^{-(k-\mu)/\beta}} \tag{1}$$

holds for all values of k. The parameters of mode (μ) and scale (β) can be estimated using minimal datapoints via the method of probability weighted moments [12]. Once the parameters of Y are known we can answer questions of the form in Equation 1: What is the probability that a student S, who did not work with any other student, has a similarity score Y that is greater than some threshold k used to report plagiarism? The Fisher-Tippett-Gnedenko Theorem should apply for any plagiarism measure—not just MOSS and TMOSS, and not just for programming assignments.

[2]An exponential tail is a reasonable assumption for the token count similarity score X_i. Notably, this assumption has a light impact on the results: If X_i has a sub-exponential tail (for example % similarity has a fixed upper limit and thus has no tail), Y will have a Reverse Weibull Distribution, which along with the Gumbel is a special case of the Generalized Extreme Value Distribution.

Table 2: Results of TMOSS (per snapshot) and MOSS (per final submission) on set of 1420 students.

	MOSS	TMOSS
HEC students	35 (2.5%)	61 (4.3%)
Runtime	0.03 hr	9.77 hr

6 RESULTS

After using TMOSS with human verification, we found 61 students (4.1% of the dataset) in the HEC student group. By contrast, we found only 35 students (2.5% of the dataset) using the traditional MOSS approach with human verification (Table 2). We run both experiments per-quarter by comparing only against peer submissions in the same course offering (in addition to the same set of online solutions across quarters).

We compare the distribution of final submission similarity scores in regular MOSS (Figure 2a) with that of the maximum similarity score over all snapshots in TMOSS (Figure 2b). In both cases, the distribution of similarity scores for the non-HEC student group (1359 students) fit a Gumbel distribution; (TMOSS: $\mu = 153.4, \beta = 37.8$; MOSS: $\mu = 118.8, \beta = 39.2$). However, the distribution of TMOSS scores for the HEC group is easily differentiable from the non-HEC group, where as the MOSS scores for the HEC group are less differentiable, further suggesting that TMOSS is a better metric for detecting students in the HEC group.

We also compare the effectiveness of two different MOSS similarity scores. From Section 3, the number of shared MOSS tokens and % match similarity can both be considered valid scoring backends as they are less dependent on snapshot length. For each similarity score, we filter the TMOSS top matches through various score thresholds k, and we compute the F1 score (a harmonic mean of precision and recall) for each resultant candidate HEC group. We find that the optimal threshold for number of tokens ($k = 367$ tokens) produces an F1 score of F1 = 0.97, compared to the optimal threshold for % match similarity ($k = 46\%$) at F1=0.58. Moreover, number of tokens performs better as a similarity score at all thresholds.

6.1 Performance analysis

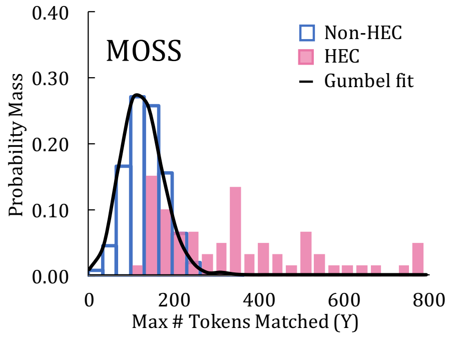
We next consider how excessive collaboration is correlated with assignment work patterns and overall course performance. Table 3 compares various groups of students characterized by their HEC traits. A quick glance at Table 3 shows almost immediately that students in the HEC group compared to those in the non-HEC group tend to start significantly later (an mean of 3.31 days before the deadline, $\delta = 2.0$ days later), have significantly lower midterm and final scores ($\delta = 0.328$, $\delta = 0.351$, respectively), spend 27% less time on task, and often attend fewer TA office hours (51% less over the entire course).

It is important to note that factors other than HEC group membership did not correlate as significantly with student exam performance. A natural one to consider is start date on the Breakout assignment. Figure 2c shows that for non-HEC students, a later

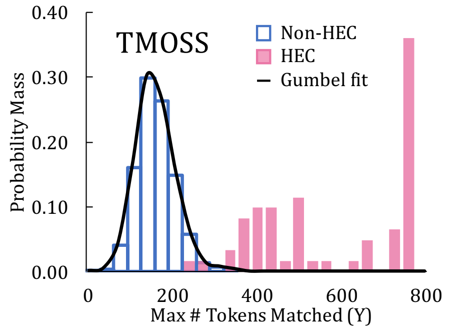
Table 3: Work patterns comparison of different student groups: (a) midterm and (b) final scores; (c) start day as number of days prior to deadline that a student started the assignment; (d) hours on task as computed in Section 3; (e) TA hours attended over the entire course.

# Students	Non-HEC 1359	HEC 61		Online-Match 55	Peer-Match 6		MOSS 35	TMOSS-only 26	
	μ (SE)	μ (SE)	$p^{‡}$	μ (SE)	μ (SE)	$p^{‡}$	μ (SE)	μ (SE)	$p^{‡}$
(a) Midterm†	.519 (.282)	.191 (.207)	**<.0001**	.192 (.208)	.183 (.202)	.50	.158 (.161)	.235 (.250)	.08
(b) Final†	.518 (.282)	.167 (.189)	**<.0001**	.169 (.193)	.156 (.148)	.49	.114 (.106)	.246 (.247)	**<.01**
(c) Start day	-5.31 (2.69)	-3.31 (2.92)	**<.0001**	-3.13 (2.95)	-5.00 (1.83)	.07	-2.97 (2.85)	-3.77 (2.94)	.14
(d) Hours on task	9.88 (4.92)	7.23 (4.34)	**<.0001**	6.98 (4.18)	9.54 (5.04)	.09	5.98 (3.65)	8.91 (4.61)	**<.01**
(e) TA hours	4.27 (9.24)	2.10 (4.04)	.02	2.03 (4.18)	2.67 (2.38)	.25	2.38 (4.61)	1.71 (3.08)	.25

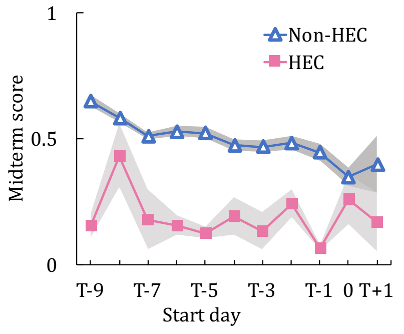
† Exam scores are exam ranking per quarter. Students who dropped the class mid-quarter were removed in each group when computing exam statistics.
‡ p-values are computed by bootstrapping for 100,000 iterations on a one-tailed hypothesis test.



(a)



(b)



(c)

Figure 2: Distribution of HEC vs Non-HEC scores by (a) MOSS and by (b) TMOSS; (c) Average exam rank with bootstrapped SE.

start on the assignment is correlated with lower performance on the midterm. However, the HEC group performs consistently lower than the non-HEC group regardless of how early or late they start the assignment. The distribution for the final exam scores were consistent with this finding; for clarity they are not shown.

Digging further into the HEC group, we compare students who exhibited excessive similarity with an online solution with those who had high scores with peer final submissions as follows: we construct an undirected connectivity graph of all the top matches of all students, with $N + n$ nodes for all students and all known online submissions, and edges connect student nodes to their top match nodes. We define an *online match* student as one whose top match is connected to an online solution in our graph; all other students are defined as *peer matches*. We find that out of the 61 students in the HEC group, only 6 students were in our peer-match group. We see that students in the online-match HEC group tend to start slightly later than those in the peer-match HEC group; however, the p-values are limited by the small sample size and we are unable to draw any real conclusions.

Finally, we consider the 26 students that were only detectable via TMOSS; these students had high similarity scores over the course of their assignment, but did not exhibit excessive collaboration in their final submission. Again looking at Table 3, we find that the student who were detectable only through TMOSS tend to perform slightly better and work slightly more than those detectable through regular MOSS, but these differences were not as significant.

7 DISCUSSION

Our results show that TMOSS, a temporal analysis of student progress to detect excessive collaboration, is both feasible and more accurate than the final-submission-based detection algorithms used today. Temporal tools are not only effective for detecting unusual patterns of students, but they can also be used to further understand work patterns as correlated with class performance in an effort to provide improved feedback.

We also found that the Gumbel distribution provides us a probabilistic interpretation for scores; this interpretation can be used in a more formal model for separating HEC from non-HEC for a future tool. It also helps absolve students' concerns of a false positive; the likelihood of getting more than 376 tokens (the similarity score threshold determined in our F1 thresholding analysis) is $\Pr(Y \geq 376) < 0.003$. We therefore believe that the likelihood of having a false positive in our results is incredibly low.

We have shown that the students who have a high match with online solutions comprise a large portion of the HEC group, and students who excessively collaborate with peers within their quarter are very low. Work remains to gauge how the HEC group would change when the analysis is expanded to compare against submissions from previous quarters. Moreover, we have yet to understand how the use of TMOSS changes student behavioral patterns. We hope that the use of similarity scores on intermediate work acts as a deterrent and helps engender an academically honest ecosystem. It would even be possible to modify our Eclipse IDE to upload intermediate work periodically, prior to final submission; this would allow us to run TMOSS over the course of the assignment and provide timely interventions for students that help them get back on track.

Finally, we emphasize that TMOSS is not a substitute for human verification; it only helps to provide a more accurate picture of the student landscape. Nor is it intended to impose a negative, pressuring environment on students; instead, we hope that TMOSS is an example of how intermediate student work can be incorporated into the overall feedback system. For instance, TMOSS can be repurposed to evaluate intermediate work on a set of unit tests; an understanding of how a solution functionally evolves over time can guide instructors in giving improved, personalized feedback for each student. We hope that improvements to tools like TMOSS will simultaneously deter plagiarism efforts and facilitate the learning process in future classrooms.

8 CONCLUSION

We have shown that TMOSS can be used to identify students who exhibit excessive collaboration with online or peer solutions and to understand student work patterns over the course of an assignment. The use of MOSS is not critical; any software similarity detector with numerical similarity scores can be used as a backend to TMOSS. TMOSS can thus also be extended to non-programming scenarios; for example, TurnItIn similarity scores [2] can be used instead to check for collaboration in essay grading.

We found that students exhibiting excessive collaboration perform worse on exams and make more limited use of TA office hours. In addition, we have found that a Gumbel distribution fits the distribution of similarity scores in the absence of any excessive collaboration.

TMOSS is just one step in the direction of designing software to better understand students in large classrooms. Such a tool separates typical students from atypical students, and can even be used to indicate at what time a student was experiencing issues. While the use of TMOSS in this work is to detect excessive collaboration, its analysis of intermediate student work can be extended to provide more timely, more accurate feedback. We can therefore identify struggling students in a large classroom, instead of waiting for them to come to us. By reaching out to these students early on, we can give them the right resources to achieve success.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-114747.

REFERENCES

[1] A. Aiken. 2014. A System for Detecting Software Plagiarism. Online. (2014). Retrieved October 25, 2016 from https://theory.stanford.edu/~aiken/moss/.

[2] T. Batane. 2010. Turning to Turnitin to Fight Plagiarism among University Students. *Educational Technology & Society* 13, 2 (2010), 1–12.

[3] R. Caldarola and T. MacNeil. 2009. Dishonesty deterrence and detection: How technology can ensure distance learning test security and validity. In *Proceedings of the 8th European Conference on E-Learning*. 108–115.

[4] CodeHS. 2016. Project: Breakout. Online. (2016). Retrieved October 25, 2016 from https://codehs.com/library/course/1/module/469.

[5] J. Forbes, D. J. Malan, H. Pon-Barry, S. Reges, and M. Sahami. 2017. Scaling Introductory Courses Using Undergraduate Teaching Assistants. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 657–658. https://doi.org/10.1145/3017680.3017694

[6] P. H. Fröhlich. 2016. Department of Computer Science at The Johns Hopkins University: 600.111: Python Scripting. Online. (2016). Retrieved October 25, 2016 from http://gaming.jhu.edu/~phf/2011/fall/cs111/assignment-breakout.shtml.

[7] D. Gitchell and N. Tran. 1999. Sim: A Utility for Detecting Similarity in Computer Programs. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99)*. ACM, New York, NY, USA, 266–270. https://doi.org/10.1145/299649.299783

[8] D. Gries, L. Lee, S. Marschner, and W. White. 2016. CS 1110: Introduction to Computing Using Python Fall 2016. Online. (2016). Retrieved October 25, 2016 from http://www.cs.cornell.edu/courses/cs1110/2016fa/.

[9] E. J. Gumbel. 1935. Les valeurs extrêmes des distributions statistiques. *Ann. Inst. Henri Poincaré* 5, 2 (1935), 115–158.

[10] E. Lau, X. Liu, C. Xiao, and X. Yu. 2004. Enhanced user authentication through keystroke biometrics. *Computer and Network Security* 6 (2004).

[11] A. Maas, C. Heather, C. T. Do, R. Brandman, D. Koller, and A. Ng. 2014. Offering Verified Credentials in Massive Open Online Courses: MOOCs and technology to advance learning and learning research (Ubiquity symposium). *Ubiquity* 2014, May (2014), 2.

[12] S. Mahdi and M. Cenac. 2005. Estimating Parameters of Gumbel Distribution using the Methods of Moments, probability weighted Moments and maximum likelihood. *Revista de Matemática: Teoría y Aplicaciones* 12, 1-2 (2005).

[13] D. L. McCabe. 2005. Cheating among college and university students: A North American perspective. *Intl. Journal for Educational Integrity* 1, 1 (2005), 10–11.

[14] N. Parlante, S. A. Wolfman, L. I. McCann, E. Roberts, C. Nevison, J. Motil, J. Cain, and S. Reges. 2006. Nifty Assignments. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)*. ACM, New York, NY, USA, 562–563. https://doi.org/10.1145/1121341.1121516

[15] A. Petersen, M. Craig, J. Campbell, and A. Tafliovich. 2016. Revisiting Why Students Drop CS1. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16)*. ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/2999541.2999552

[16] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. 2012. Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, NY, USA, 153–160. https://doi.org/10.1145/2157136.2157182

[17] J. Pierce and C. Zilles. 2017. Investigating Student Plagiarism Patterns and Correlations to Grades. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 471–476. https://doi.org/10.1145/3017680.3017797

[18] L. Prechelt, G. Malpohl, and M. Phlippsen. 2000. *JPlag: Finding plagiarisms among a set of programs*. Technical Report. Universitat Karlsruhe.

[19] E. Roberts, J. Lilly, and B. Rollins. 1995. Using Undergraduates As Teaching Assistants in Introductory Programming Courses: An Update on the Stanford Experience. In *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '95)*. ACM, New York, NY, USA, 48–52. https://doi.org/10.1145/199688.199716

[20] S. Schleimer, D. S. Wilkerson, and A. Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 76–85. https://doi.org/10.1145/872757.872770

[21] J. Schneider, A. Bernstein, J. vom Brocke, K. Damevski, and D. C. Shepherd. 2016. Detecting Plagiarism based on the Creation Process. *CoRR* abs/1612.09183 (2016). http://arxiv.org/abs/1612.09183

[22] J. G. Spohrer and E. Soloway. 1986. Analyzing the high frequency bugs in novice programs. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. Ablex Publishing Corp., Norwood, NJ, USA, 230–251. http://portal.acm.org/citation.cfm?id=28897

[23] G. Whale. 1988. Plague: Plagiarism Detection Using Program Structure. In *Tech. Rep. 8805*.

[24] M. J. Wise. 1996. YAP3: Improved Detection Of Similarities In Computer Program And Other Texts. In *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*. ACM Press, 130–134.

[25] J. Young. 2012. Dozens of plagiarism incidents are reported in Coursera's free online courses. *The Chronicle of Higher Education* (Aug 2012). http://www.chronicle.com/article/Dozens-of-Plagiarism-Incidents/133697