ASEE'S VIRTUAL CONFERENCE
At Home with Engineering Education JUNE 22 - 26, 2020 #ASEEVC
Paper ID #28368

Misunderstandings, mistakes, and dishonesty: A post-hoc analysis of a large-scale plagiarism case in a first-year computer programming course

Dr. Philip Reid Brown, Rutgers, The State University of New Jersey

Philip Brown is an Assistant Teaching Professor in Undergraduate Education at Rutgers School of Engineering. Philip recently received his PhD from the Department of Engineering Education at Virgnia Tech. His research interests include the use of motivation, cognition and learning theories in engineering education research and practice, and better understanding student perspectives in engineering programs.

Dr. Ilene J. Rosen, Rutgers, The State University of New Jersey

IIene Rosen has been an educational administrator serving students in higher education for 35 years. She earned her doctoral degree in educational psychology from Rutgers University Graduate School of Education. Currently the Associate Dean for Student Services at Rutgers, School of Engineering, she also served as the director of several programs including the NJ Educational Opportunity Fund Program at Rutgers School of Engineering, the NJ Governor's School of Engineering & Technology, and the Northern NJ Junior Sciences Symposium. Rosen has been recognized as the Educator of the Year in Higher Education by the Society of Hispanic Professional Engineers.

Misunderstandings, mistakes, and dishonesty: A post-hoc analysis of a large-scale plagiarism case in a first-year computer programming course

Introduction

In this evidence-based practice paper, we discuss the issue of plagiarism in a first-year engineering computer programming course. Plagiarism is an issue that can plague any course that asks students to submit independently created work. Traditionally, plagiarism has been associated with writing assignments, and there are a wide variety of tools and interventions available for both identifying and preventing plagiarism on these assignments. However, although computer programming courses also report a large number of plagiarism cases, there are fewer easy to use or well understood tools and interventions available to instructors of these courses. This paper describes a sequence of plagiarism cases in a large first-year computer programming course for engineers, and how the course was adapted in order to address the prevalence of these cases.

Part of the issue with plagiarism in computer programming is a lack of consensus on what is and is not ethical to copy and use without acknowledgement when it comes to computer code. Many programmers gladly share code openly, and being able to find examples of code that can help you write a program can be a valuable and valid skill for a programmer. However, when courses are tasked with teaching and assessing the basic principles of computer programming, there is a dissonance between the free-sharing, open culture often found in some programming communities, and the needs of instructors when it comes to determining that students understand those basic principles. Additionally, we often encourage students to work in groups (and group work can be a boon to motivation, engagement, and learning) in engineering courses [e.g. 1], which can sometimes lead to confusion about the limits of plagiarism when submitting individual work. Some computer programming courses may avoid plagiarism by focusing on closed book testing for assessment. However, in addition to the universally acknowledged drawbacks to test-centric assessment[2],[3], the knowledge displayed in test answers is a less authentic representation of computer programming skill than projects that ask students to write and test real computer programs.

To combat plagiarism, project-centric programming courses often use plagiarism software like Stanford MOSS in order to flag and investigate potential plagiarism cases. The idea behind the use of such a program is that of deterrence: If these tools are good enough at detecting plagiarized code, and students are aware of their existence, then students will not plagiarize, lest they get caught with solid, algorithmic proof behind the potential accusation. In practice, it is not so simple. Some students attempt to beat plagiarism programs. More difficult still is when students are not aware of their own plagiarism, or when students work together and wind up with similar code. As discovered in the course discussed in this paper, plagiarism detection does not solve the problem of plagiarism, it merely confronts it, along with all of the gray areas that surround it. In the right context, this confrontation could be beneficial. However, we found that it simply produced an array of additional time and energy-intensive problems.

The result of these problems was the realization that a different approach was needed in order to curb plagiarism: one that circumvents and dis-incentivizes direct plagiarism while allowing students the freedom to use once “gray” areas such as working closely together. Our strategy involves assigning projects as normal, while using open note and open computer quizzes to assess those projects instead of direct submission of code. We will discuss the process we went through in creating this new assessment strategy, as well as the strategy itself, throughout the rest of this paper. Note that this paper is not a traditional research paper. The goal of this paper is to give the reader an understanding of why certain course design decisions were made, and provide some insight into the potential underlying causes of plagiarism in computer programming courses.

What do we know about plagiarism?

Plagiarism is the theft or unauthorized use of intellectual property. While traditionally thought to be at home in the realm of writing, computer programmers have long acknowledged the potential for high prevalence of internet-assisted plagiarism in programming classrooms [4]. More recent research suggests that up to 10% of computer programming assignments might contain plagiarized code, though such results vary from assignment to assignment [5].

Research outside of computer programming suggests that students have a wide range of reasons for plagiarizing, ranging from laziness, to a desire to help friends (in cases of facilitation of plagiarism) [6]. However, where students might traditionally see plagiarism as immoral or an action to be avoided when it comes to writing, studies suggest that students might hold a different perspective with regards to computer programming, and may not always see actions that may be considered plagiarism as wrong [7]. While one option may be to educate students on what is considered plagiarism in computer programming courses, such interventions have had mixed effectiveness [5], [8]. In reality, plagiarism is likely to be a persisting problem in computer programming courses. Given what we know about plagiarism in general, we will now describe how it has affected the programming course at the center of this paper.

Course Description

The course described in this paper is a large (400-700 student), first-year programming course at a large, land-grant university in the Mid-Atlantic United States. The programming course teaches the MATLAB programming language, and focuses on the fundamentals of computer programming for student learning objectives. The course covers topics including logic, binary, variables, data types, user-defined functions, conditional statements, loops, data visualization, data processing, data analysis, and engineering applications.

The current format of the programming course has a lecture/recitation format, with 3 lectures of 100-250 students and 14-16 recitations of 20-44 students. Teaching strategies focus on student activities and hands-on, active learning, although some traditional lecturing techniques are used in large lecture meetings out of necessity. Lectures and recitations meet once a week, each, and both are 80 minutes long. Lectures consist of 15-20 minute blocks of more traditional lecture, where new information is presented, interspersed with activities. Recitations consist almost entirely of hands on activities and projects that follow up on the previous week’s lecture topics.

Homework and reading are assigned online, via an online textbook platform. There are also two midterm exams and one final exam, which account for approximately 50% of the course grade in total. Individual projects and quizzes account for 20% of the course grade. The remaining grade distribution has evolved over the past few years, but has been some combination of team-based final projects (now no longer in use), homework, in-class assignments, attendance, and participation.

Instructional staff consists of 1 faculty member, who teaches all lecture sections and sometimes teaches recitations, 7-8 graduate student instructors who teach recitations, and 15-20 undergraduate teaching assistants who help facilitate recitation activities.

The course has undergone significant redesign over the past 4 academic years, starting with a complete overhaul and redesign in the Fall of 2016 [9]. Since then, various changes have been made to the course, including the addition of an online textbook and the introduction of larger projects in addition to smaller homework and recitation assignments. The percentage of students who must retake the course due to poor performance or withdrawal has remained lower than pre-redesign levels, as one goal of the redesign was to stop the course from being a "weed out" course. However, there are notable fluctuations from semester to semester.

Semester	**Total Students Who Had to Retake Course***	**Initial Course Enrollment**	**Percentage**
Fall 2015**	123	573	21.5%
Spring 2016**	135	530	25.5%
Fall 2016 #	34	596	5.7%
Spring 2017	80	611	13.1%
Fall 2017	27	621	4.3%
Spring 2018***	63	505	12.5%
Fall 2018	65	672	9.7%
Spring 2019 $	66	449	14.6%
Fall 2019 $	56	624	9.0%

* Students who withdraw or receive a D or F must retake the course to remain in engineering
** Prior to current course design *** Individual projects introduced
# Prior to use of online text or projects $ Semesters with quizzes for project assessments

Table 1: History of Enrollment and Performance in Programming Course

Table 1 shows the history of student performance outcomes from the past 5 years of the course. One notable trend from before and after the redesign is that more spring semester students perform poorly or withdraw from the course than fall semester students. The co-authors of this paper and course instructors have theorized about possible reasons for this. Our main theory is that the population of students in the fall are almost entirely true first-year students, while the population in the spring is a mixture of first-year and transfer students, as well as students retaking the course, and that students from these distinct groups have different outcome likelihoods.

In general, the introduction of individual projects has coincided with a slightly higher number of students performing poorly, overall. We theorize that, among other possible reasons, the introduction of official plagiarism checking protocols while assessing individual projects, and the issues that result in some students plagiarizing, are an influence on higher rates of poor performance.

Course Projects and Plagiarism

Throughout the first two years of the course redesign, students were required to submit smaller, weekly coding assignments in the form printouts or online uploads of code. The idea behind these submissions, in addition to some automatically graded online homework, was to give students detailed, individualized feedback on their code.

Faculty and graduate instructors noticed a high number of identical or similar submissions on these assignments. While plagiarism checks were considered, they were not widely implemented due to a lack of resources. Though applications like Stanford MOSS are excellent at flagging submissions for potential plagiarism, such flags are the beginning of a long process that includes human confirmation of potential plagiarism, and collecting and formatting plagiarism reports for potential academic integrity submissions. Graduate instructors were already reporting heavy weekly grading loads with exams and homework, and course faculty did not have time to sift through potentially hundreds of plagiarism flags per week. Reporting was reserved for the most egregious cases of identical submissions.

However, as some assignments were nullified due to the large number of identical submissions, we realized that there was a need for a new format of assignment where individual feedback could be given, and where we could also hold students accountable for doing their own work. Starting in the spring of 2018 we began assigning individual projects 3-4 times a semester. To compensate for this added work load for students and instructors, individual homework assignments were moved to being completely online and graded automatically, and were slightly reduced in number.

Individual projects were designed to tie together multiple concepts from throughout the semester, and progress in complexity through the semester as more concepts were introduced to the course. Though we do not reuse project prompts from semester to semester, there are many similarities in the concepts covered by projects from semester to semester. Project 1 usually asks students to create a user-defined function that can choose from an assortment of mathematical equations to perform. Project 2 usually asks students to write a function that processes 1-dimensional arrays using loops, and sometimes conditional statements. Projects 3 and 4 involve a variety of topics related to engineering applications, usually involving some combination of multi-dimensional arrays, multiple data types, and multiple user-defined functions in addition to data analysis and visualization. Appendix A includes an example Project 2 from the fall 2019 semester.

In assessing individual projects, we wanted to see whether students were able to both understand and implement various concepts, and apply them to practical applications in engineering. Initially, we assessed submissions of code. Code had to be fully commented, with descriptions of how different programming methods worked and why they were used. Grades were assigned both on correctness, style, and the use of appropriate or required programming methods.

In an effort to curb plagiarism, students were repeatedly informed of the use of anti-plagiarism software (Stanford MOSS) on project submissions. The first project of the spring 2018 semester was excluded from this process due to the relatively small size and simple implementation of the project. However, using Stanford MOSS, the course faculty found evidence that a large number of students were sharing code or copying code on Project 2 and Project 3 (see Table 2). Note that Table 2 only has numbers for students who were officially reported for plagiarism, and that there were some students reported multiple times each semester.

When using Stanford MOSS, we used an 80% similarity as the cut-off for a potential plagiarism flag. Any two submissions containing that level of similarity or higher were then inspected by the course faculty. If the faculty thought the similarities were likely due to copying or improper sharing, the student was given an opportunity to meet one on one to discuss the similarities. Students who were thought to have plagiarized after this process were given a 0 on the assignment and reported to central offices responsible for student conduct.

Many potential cases were dismissed without being reported, as some gray areas were quickly identified. What should be done with students who worked closely together, but put in significant amounts of work and clearly understood the code they submitted? What happens in complicated situations where some students shared a modest number of ideas, but then one of those students shared a complete program with an additional student? While instructions did say to work individually, *it was clear that those instructions were potentially untenable within the reality of the course.*

Semester	**Plagiarism Cases Reported**
Spring 2018	74
Fall 2018	138

Table 2: Plagiarism Cases in Programming Course

In the following semester, fall 2018, efforts were made to clarify expectations of what constituted plagiarism in these assignments. Students were allowed to work together, *so long as they reported who they worked with in a comment* and did not submit code that was identical. Comments played a big role in determining what was acceptable: students with very similar code but descriptive comments clearly written in their own words were not reported for plagiarism, while students with similar or identical comments were reported. There were repeated reminders about how to approach projects, *including a one-page guide* of recommendations that was included with each project. However, rates of suspected plagiarism from the Stanford MOSS tool were still high, and more of the "gray area" cases were reported due to evidence that students were clearly ignoring or disregarding specific instructions about how to avoid plagiarism.

It was clear that we could not possibly continue to administer these assignments as such. From an instructor's point of view, processing potential plagiarism was taking up an inordinate amount of time, and took time away from pursuing tasks that are more conducive to course improvement and student learning. However, before altering these assignments, we needed to understand what some of the underlying issues at play were. Examining notes from meetings with students flagged for plagiarism, as well as observations from faculty, graduate, and undergraduate instructors, we first tried to understand why students were making choices that lead to plagiarism.

Why did students plagiarize?

To understand the potential causes of plagiarism, it is first important to understand the different kinds of work that might get flagged for plagiarism. Tools like Stanford MOSS are simply measures for similarity, and there are many different reasons *why* two pieces of code might score highly in similarity. Table 3 breaks down three general categories: Those that are not plagiarism, those that fall in a gray area, and those that are definitely plagiarism. Please note that we will not reference specific cases, even with pseudonym, so as not to risk identifying students. Instead, we will discuss generalities, and when necessary, hypothetical cases that may be amalgamations of details observed while processing plagiarism cases.

Category	**Example**	**Result with Project Submission**	**Result with Quiz Submission**
Not Plagiarism	Students use a similar obvious, concise method	Potentially flagged by Stanford MOSS, but no action taken.	Student likely passes quiz.
	Students talk to each other and share some ideas.		
	Students give each other feedback on written code.		
Gray Area	Student gets tutor-like help from another student who completed the project.	Likely flagged by Stanford MOSS, potentially reported as plagiarism depending on level of similarity	Students given opportunity to demonstrate knowledge and understanding: those who did work pass, those who did not likely fail.
	Two students work very closely together, sitting side by side while writing code.		
	A community of students shares a significant number of ideas.		
Plagiarism	A student provides another student with code as a “reference,” and that reference is copied.	Almost certainly flagged, reported as plagiarism.	The student likely fails the quiz
	A student retrieves code from an online resource like Chegg or MATLAB Central and uses that code.		
	A student steals code from another student.		

Table 3: Examples of types of submissions that could be flagged for plagiarism

Not plagiarism

In interviewing students flagged for plagiarism, there are some cases that, after examination, reveal themselves to not be plagiarism. Often, when students find *concise* ways to write programs, or portions of programs, they might stumble upon a sequence of code that others are also likely to come up with. Upon inspection, these programs are often formatted using slightly different style, and with clearly different comments. Other causes for similar situations might include academically honest communication: sharing ideas without directly sharing code could theoretically result in similar programs, as could providing feedback on another students' code, especially if changes are suggested that lead to similar strategies. These are not reasons for plagiarism, but rather, reasons for false alarms. While they are easier to deal with than the other examples in Table 3, they still take up time and energy to resolve.

Gray areas

The second, broad category of cases are the "gray" areas: Those that, depending on the perspective or details that have been revealed, can be thought of as acceptable, or academically dishonest. These cases have the greatest variety in rationale behind them, but generally center upon what "acceptable" collaboration is. For instance, we ran into many cases where groups of students had very similar code. Upon speaking to parties involved, the following story might evolve:

- Student A and B worked together closely, but have clearly different submissions via comments and style, and report each other as collaborators.
- Student C, in a panic, asks Student B for help after Student B submitted their project. Their submission is very similar to Student B, and thus Student A, but they do not report working with Student A.
- Student C finish up their code while working with Student D. They wind up with nearly identical programs. Student D forgets to include collaborators in their comments.

Upon interviewing all of these students, it may be clear that few if any of them may have intended to be academically dishonest. Nevertheless, depending on the similarity of code, and how they shared it with one another, some of them *may* have plagiarized or facilitated plagiarism. Nevertheless, also dependent on the nature of each interaction, each student may have developed an understanding of how to accomplish the project. While not always ideal ways to learn, these gray areas may not be things we want to discourage. We *want* students to build connections with peers as learning resources, especially in large courses where other personal resources might be scarce. However, code submission assessments do not give us adequate ways of determining the difference between students who learned in this situation, and those who simply copied an answer.

Students in these gray areas often discussed their peers as being the most convenient resource available to them, while also being in need of additional learning resources. They also often reported being confused about what was and was not acceptable collaboration. Though we provided specific, detailed instructions for how to approach these assignments, and the possibility of working together, it was clear that these often got lost in the mix as students focused, first, on how to complete the assignment. Many students reported, true or not, that they forgot or did not realize that they were supposed to report collaborators. It is possible that, in providing so many details of what to do and not to do in these assignments, the cognitive load was too much for students who may have also been expending a great deal of energy on learning a potentially difficult subject. While our individual expectations of students may have been reasonable, the overall collection of them may have been too much in some cases.

Plagiarism

Our final broad category is that of cases that are clearly plagiarism or facilitation of plagiarism. While these might be easy to identify, they can be harder yet to understand. Why do students risk penalties beyond a 0 on an assignment in order to pass somebody's work off as their own? Why do some students choose to give their work to another, knowing that person could easily copy it?

Despite the obviousness of the cases, some students in these situations were still confused about the bounds of plagiarism in computer programming. Many insisted that there was only one or two ways to write a given program, despite evidence that their program was only a match with one other student in a class of 600. Many students said they were simply trying to give or receive help on an assignment, and did not intend to commit plagiarism. Still others admitted to being desperate, and feeling like they had no other way of completing assignment.

Semester and Assessment	**Project**	**N**	**Pass**	**Fail**
Spring 2018 Submit Project Code	1	453	421 (93%)	32 (7%)
	2	453	339 (75%)	114 (25%)
	3	453	357 (79%)	96 (21%)
Fall 2018 Submit Project Code	1	600	573 (96%)	27 (4%)
	2	600	378 (63%)	222 (37%)
	3	600	406 (68%)	194 (32%)
	4*	600	203 (34%)	397 (66%)
Spring 2019 Quiz Assessment	1	431	206 (47%)	225 (53%)
	2	431	129 (30%)	302 (70%)
	3	431	161 (37%)	270 (63%)
	4	431	271 (62%)	160 (38%)
Fall 2019 Quiz Assessment	1	599	557 (93%)	41 (7%)
	2	599	407 (68%)	192 (32%)
	3	599	423 (71%)	176 (29%)
	4	599	419 (70%)	180 (30%)
*In the Fall of 2018, students knew that one project would be dropped. Many students did not submit Project 4 for this reason.				

Table 4: Project pass and failure rates before and after quiz assessments.

Looking at the numbers on a whole, as seen on Table 4, it's clear that there were more students who failed their project submissions than there were students who plagiarized. While Table 4 does not show this information, **the vast majority of project failures in the Spring and**

Fall of 2018 were through non-submissions, and the vast majority of passing project grades received nearly full marks. For every student that gave up on a project and plagiarized, there were two or three students who gave up and did not submit anything.

This is a stark, but not unexpected, situation. Computer programming courses, more than almost any other family of courses, have a distinct issue of differences in preparation between incoming students. In an earlier study of this course, we found that approximately half of the class had programmed before, while the other half had not [10]. While we designed this course with non-programmers in mind, it can be difficult to gauge the needs of the most at-risk students in a course when they are not necessarily the loudest voices. In that previous study, it was found that students with no prior programming experience had worse outcomes than students with prior programing experience. As we discovered in subsequent semesters with quiz assessments, many of these students were not learning essential skills such as **running their own programs** and **testing code** because project submissions, and likely homework assignments before them, were not guaranteeing that those skills were being assessed. Excepting facilitators of plagiarism, the vast majority of students reported for plagiarism were students with no prior programming experience.

It was with these considerations that we went into the following semester with the idea that we would combat plagiarism by making it a non-factor in assessment. The results were, at first, not as positive as we had hoped.

Circumvention and Prevention

In the Spring of 2019, we decided to keep our individual projects, but change the way they were assessed. Instead of having students submit project code, we instead required students to take a 30-minute open note, open computer quiz to demonstrate their completion of the project.

Quizzes are administered in an active learning style classroom, with large, round tables. The capacity if the classroom is approximately 72 students, but sections never have more than 45 students, allowing for adequate spacing. Quizzes were proctored by one graduate student instructor and two undergraduate teaching assistants. To attempt to prevent unauthorized access, quizzes were password protected and time limited. To curb cheating, questions with numerical answers had multiple, randomized versions administered in each quiz session. While students could use computers, phones and chat-related computer apps were forbidden, and proctors were mindful to check for inappropriate applications on computer screens.

The format of these quizzes has not evolved much since their inception, and is exemplified by Appendix B. Students are given a selection of short answer questions via our course management system’s quiz application. These questions take the following format:

- Asking that students use their programs with specified inputs, or as part of pre-written scripts, and report the result.
- Asking students to make a minor modification to their program, and show us the entire modified program or just he code that was added or modified.

- Asking students to describe how certain code works, or which programming methods they chose in accomplish a certain task.

Through these questions, the hope was that students who completed the project would have no trouble getting full credit, assuming that the project was correct and they understood how it worked, while students who attempted to complete the project dishonestly would not be able to answer questions in the time allotted. While we believe that this is what occurred, Table 4 shows that *a very large portion of students* were unable to answer most quiz questions correctly, suggesting that most students were not completing the projects to the level that we were expecting.

Repeated efforts were made throughout the semester to make students aware of how the quizzes work, and the types of questions they should be prepared to answer. The quiz format did not change significantly, but quiz results, for the most part, did not improve throughout the semester. One cause of this could be the atmosphere of the class that semester. Many students were upset that, after their colleagues were allowed to submit code the previous semester, they had to take quizzes for the same assignments. Some were adamant that we were making the class more difficult for difficulty's sake, and continued to answer quiz questions by copying and pasting project code throughout the semester. While we tried to be clear about our reasons behind our project assessments, and our intention to give projects that were roughly equivalent to the previous semesters', the damage was done and the mood of the course did not improve.

There are many potential reasons for the atmosphere encountered in the spring of 2019. As mentioned previously, the population of students in the spring semesters may be less prepared than fall semesters, overall. Some of it could be attributed to personality clashes between students and some instructors. However, we believe the biggest cause of these problems was that **quizzes were assessing essential skills that previous assessments were not.** Specifically, students struggled with using their own code to answer questions, and performing tasks that required them to run code that we prepared for them. While these topics are a focus of the early portion of the course, they may not have been as essential to completing previous assessments, many of which called and tested student code for them. Students also struggled with describing how code worked, and making simple changes such as changing the parameters of a conditional statement (i.e. deciding to check the value of two pieces of data instead of one). We believe that, when students experienced these failures at the beginning of the semester, a large portion of them "gave up" on the course, and simply attempted to secure a passing grade, rather than learn.

Without being fully aware of the reasons why students struggled on quizzes, we wound up making the decision to significantly deemphasize projects in final grade calculations, as we were unsure whether they were fair assessments at that point. Over summer break we were able to re-evaluate what the quiz results meant. Quizzes pointed to a lack of basic programming knowledge, so we redoubled our efforts to teach and reinforce that knowledge in the subsequent semester. The results (again, shown in Table 4) demonstrate an improvement in quiz outcomes **without a significant alteration of quiz format.** Adjustments to course curriculum were also minor, but clearly necessary. Greater emphasis was placed on how to test code, including code that uses different programming methods, and additional instructions were included in project documents about how to test code (See Appendix A).

Discussion and Conclusions

Throughout the process of first combatting plagiarism, and then circumventing it, we arrived upon some potentially useful take-a-ways. First, plagiarism is a symptom, not a disease. Second, many programming assessments may not guarantee that essential programming skills are assessed, even if those skills seem trivial. Finally, initial failures of new assessments can still be useful within the iterative process of course assessment. We would also like to acknowledge the limitations of what is discussed in this paper.

Plagiarism is a symptom

When we first approached the issue of plagiarism in our computer programming course, it was easy to take the viewpoint that plagiarism was the fulcrum of the battle we are fighting. And it is easy to see why: it can be infuriating that people blatantly copy the work of others for personal gain, especially when it occurs on such a large scale. It is also pervasive at many different levels in computer programming course, a fact that has even received national media attention [10]. However, the focus on plagiarism hides a host of underlying issues. In our course, we encountered gray areas where the enforcement of plagiarism rules and student learning may not be in alignment. We also discovered that plagiarism cases were mostly among students with no prior experience, suggesting that plagiarizing could be as much a last-ditch attempt to salvage a lost situation, rather than a devious plan to break rules.

This is not to suggest that we should scale back efforts to detect plagiarism or report individuals who commit it. However, we should also be understanding of what the underlying issues might be. Previous research has suggested [5], [8], and we have experienced in teaching this course, that it can be difficult to educate students on what constitutes plagiarism in computer programming classrooms as a means of preventing future plagiarism. In greater likelihood, plagiarism will remain a problem as long as the underlying causes do.

The basics can be hard to assess

The first two years of our course had a wide variety of assessments: hands-on group work, exams, auto-graded and hand-graded homework assignments, and in-class activities. However, while we had some idea that some students were struggling with understanding how to call their programs, or run code that was provided for them, we did not realize how pervasive this problem was until we introduced project quizzes. Thus, an array of assessments, many of which would seem to *imply* a basic skill, have the potential to produce false positives for the presence of that skill. We would recommend that other introductory programming courses consider this possibility.

Failed assessments are not always failures

When we first began assessing projects with quizzes, we were taken aback by the sheer chaos that we encountered. Students were not happy, and they were not able to answer questions we considered trivial. Our first, and possibly second and third inclinations were that there was something seriously wrong with the format of the quizzes. However, with some hindsight, we realized that the quizzes were simply highlighting something that was seriously lacking in the course: a significant focus on running and testing code. Again, this seemed like something that students should have been able to do given other assessments, but we were wrong. Large courses and limited one on one time with students can, clearly, hide large problems.

Limitations

This paper is not a traditional research paper. The data discussed within was collected for assessment purposes, and in some cases, was not in an ideal format for reporting. Interviews for plagiarism cases were not collected for research purposes, and thus could only be summarized in general terms.

We also realize that the assessment suggested in this paper may not be possible to easily implement for every programming class. Open-computer assessments carry the risk of other types of academic dishonesty, require universal access to computers, and can be difficult to proctor. While we are confident that we deter most students from attempting to cheat in our quizzes, you may not be as confident. It should also be noted that project quizzes do not necessarily decrease the time spent grading projects, and thus may require resources not available to other programming courses.

Conclusions

We believe this paper highlights some of the issues related to detecting and preventing plagiarism in computer programming courses. As plagiarism continues to be a key issue in the assessment of computer programming ability, we believe that a deeper examination of the causes of plagiarism, and potential solutions outside of plagiarism detection, should continue to be examined. We also believe the project quiz assessment strategy may be a viable strategy for similar programming courses.

References

[1] Terenzini, Patrick T., et al. "Collaborative learning vs. lecture/discussion: Students' reported learning gains." *Journal of Engineering Education* 90.1 (2001): 123-130.

[2] Vitasari, P., Wahab, M. N. A., Othman, A., Herawan, T., & Sinnadurai, S. K. (2010). The relationship between study anxiety and academic performance among engineering students. *Procedia-Social and Behavioral Sciences*, *8*, 490-497.

[3] Bell, A. E., Spencer, S. J., Iserman, E., & Logel, C. E. (2003). Stereotype threat and women's performance in engineering. *Journal of Engineering Education*, *92*(4), 307-312.

[4] Joy, M., & Luck, M. (1999). Plagiarism in programming assignments. *IEEE Transactions on education*, *42*(2), 129-133.

[5] Daly, C., & Horgan, J. (2005). Patterns of plagiarism. *ACM SIGCSE Bulletin*, *37*(1), 383-387.

[6] Devlin, M., & Gray, K. (2007). In their own words: A qualitative study of the reasons Australian university students plagiarize. *High Education Research & Development*, *26*(2), 181-198.

[7] Aasheim, C. L., Rutner, P. S., Li, L., & Williams, S. R. (2019). Plagiarism and programming: A survey of student attitudes. *Journal of information systems education*, *23*(3), 5.

[8] Le, T., Carbone, A., Sheard, J., Schuhmacher, M., de Raath, M., & Johnson, C. (2013, March). Educating computer programming students about plagiarism through use of a code similarity detection tool. In *2013 Learning and Teaching in Computing and Engineering* (pp. 98-105). IEEE.

[9] Brown, P. R. (2017, June). Work in progress: From scratch-the design of a first-year engineering programming course. In ASEE Annu. Conf. Expo. Conf. Proc.

[10] Bidgood, J., & Merrill, J. B. (2017). As Computer Coding Classes Swell, So Does Cheating. *New York Times*, A1.

Appendix A – Example Project

Project #2 – Sifting Data and Calculating Statistics

Oftentimes in engineering, and in other applications, we collect raw data that needs to be sorted into different groups, or ***strata*** before we can process it and make use of it. For example, if an engineer were assisting in a maintenance project on an existing bridge, they might be tasked with taking measurements of any deformations on any load bearing joints. When collecting data, she may take measurements of different joints (joints connecting sections of roadway, joints connecting trusses, joints connecting support beams, etc.), and it may be most convenient to store all of that data together during the collection process.

Once the data collection process is over, however, the engineers performing analysis on this data might need to sort it into different categories. One team might be working on designs for retrofitting roadway joints, for example, and might only care about the data collected for those joints.

In this project, you will be writing a function that will take raw data in the form of arrays, and return the subset of that data from a desired category, along with statistics about that data. ***Please note that while I used bridge joint data as an example, the function you are writing will apply to any generic raw data that needs to be sorted.***

Inputs and Outputs

Inputs

Your function will take **three** inputs:

- `rawData` – An array of doubles – This is the array that contains all of the raw numerical data that was collected, from which you will select the desired subset of data.

- `categories` – An array of doubles – This array contains category labels that correspond do the elements in the same indices of `rawData`. For example, somebody may label four different categories of data with the numbers 0, 1, 2, and 3. In this case, the `rawData` input might look like this:

0.671	-1.207	0.717		1.630	0.489	1.035	0.727	-0.303	0.294	-0.787

  While the `categories` input might look like this:

0	3	2	0	2	1	3	3	2	1

  Note that **the categories above** are just an example. **You should not assume a specific number of categories** in your `categories` input, nor should you assume that they will be in a specific format (i.e. integers).

- `selectedCategory` – A double – This is the category of data that you are sifting out of `rawData`. In the above examples, if we were to use 0 for `selectectedCategory` input, we would be looking to sift out the numbers 0.671 and 1.630 from `rawData`. **Again, as both the categories and the category to select are both inputs, you should not make any assumptions about the format of either except that they will be of the Data Type double.**

Outputs

Your function will have four outputs:

- `siftedData` – An array of doubles – This will be the elements of `rawData` for which the elements in the corresponding indices of `categories` match the value of `selectedCategory.`

- `siftedMean` – A double – This will be the mean (average) of all the values in `siftedData`.

- `siftedStd` – A double – This will be the standard deviation of all the values in `siftedData.`

- `siftedNorm` – An array of doubles – This will be the data from `siftedData` normalized so that its mean is 0 and its standard deviation is 1. **This conversion is performed by subtracting the mean and dividing by the standard deviation.** Often, when we are making decisions with multiple dimensions (i.e. sources) of data, it is necessary to normalize all data before using any data analysis tools.

Methods

You will need to consider the following programming methods in implementing this project:

- Using loops to index arrays.
- Creating multiple counting variables to track different things:
    - The indices of `rawData` and `siftedData` will be different!
- Using conditional statements inside of loops.
- Calculating statistics (use of MATLAB built in functions to do this is OK.
- Performing calculations with the elements of arrays.

Testing Code

To test your code, we recommend **generating a collection of matching `rawData` and `categories` arrays.** These do not need to be incredibly long. Try this out with different numbers of categories, and try to select different categories from the same arrays. **As always, we recommend using scripts for your tests so that you can modify them and perform them again as you need.**

Appendix B – Example Quiz Questions

Note – All quizzes are open note and open computer.

1. Please copy and paste the following code into a script in MATLAB, replacing "yourFunction" with the name of the function you wrote for this project.

```
rng('default')
rng(2)
data = randn(30,1)*5+2;
categories = ceil(rand(30,1)*3);
selectedCategory = ceil(rand*3);

[~, answer1, answer2, ~] =
yourFunction(data,categories,selectedCategory);
```

What are the values of answer1 and answer2? You only need to include up to 4 decimal places in scientific notation.

2. Please copy and paste the following code into a script in MATLAB, replacing "yourFunction" with the name of the function you wrote for this project.

```
rng('default')
rng(1)
data = randperm(9);
categories = ceil(rand(9,1)*3);
selectedCategory = ceil(rand*3);

[answer1, ~, ~, answer2] =
yourFunction(data,categories,selectedCategory);
```

What are the values of answer1 and answer2? You only need to include up to 4 decimal places in scientific notation.

3. In 2-3 sentences. Please discuss how you implemented loops, counting variables, and array indexing in this project.

4. Assume that, in addition to the conditions described in the project document, we only wanted to accept numbers from selectedCategory with a magnitude (i.e. absolute value) greater than 0.5 into siftedData. Any numbers with lower magnitude in rawData of selectedCategory would be ignored. Please update your code to accomplish this, and paste your entire program as the answer to this question.

5. In 3-4 sentences, describe the code you had to edit in the previous question to accomplish the desired changes. Was it inside of any other programming structures? Why or why not?