MINING STUDENT SUBMISSION INFORMATION TO REFINE PLAGIARISM DETECTION

A Thesis

by

KATE ASHLEY CATALENA

Submitted to the Office of Graduate and Professional Studies of

Texas A&M University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Dilma Da Silva
Committee Members,	Krishna R. Narayanan
	Frank M. Shipman III
	J. Michael Moore
Head of Department,	Scott Schaefer

May 2020

Major Subject: Computer Science

ABSTRACT

Plagiarism is becoming an increasingly important issue in introductory programming courses. There are several tools to assist with plagiarism detection, but they are not effective for more basic programming assignments, like those in introductory courses. The proliferation of auto-grading platforms creates an opportunity to capture additional information about how students develop the solutions to their programming assignments. In this research, we identify how to extract information from an online autograding platform, Mimir Classroom, that can be useful in revealing patterns in solution development. We explore how and to what extent this additional information can be used to better support instructors when identifying cases of probable plagiarism. We have developed a tool that takes the raw student assignment submissions from Mimir, analyzes them, and produces data sets and visualizations that help instructors to refine information extracted by existing plagiarism detection platforms. The instructors can then take this information to further investigate any probable cases of plagiarism that have been found by the tool. Our main goal is to give insight into student behaviors and identify signals that can be effective indicatives of plagiarism. Furthermore, the framework can enable the analysis of other aspects of students' solution development processes that may be useful when reasoning about their learning.

As an initial exploration scenario of the framework developed in this work, we have used student code submissions from the CSCE 121: Introduction to Program Design and Concepts course at Texas A&M University. We experimented with the student code submissions from the Fall 2018 and Fall 2019 offerings of the course.

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Da Silva, for her guidance and support throughout this research. I would also like to thank my committee members, Dr. Shipman, Dr. Moore, and Dr. Narayanan, for their guidance and insight throughout this research.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Dr. Dilma Da Silva of the Department of Computer Science and Engineering and Dr. Frank Shipman of the Department of Computer Science and Engineering, Professor J. Michael Moore of the Department of Computer Science and Engineering, and Dr. Krishna R. Narayanan of the Department of Electrical and Computer Engineering.

Funding Sources

This graduate study was supported by internal faculty funding.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

1. INTRODUCTION

In the last few years, it has become evident that plagiarism in introductory programming courses is more and more prevalent. A New York Times article, published May 29, 2017, indicated that as the demand for computer science education increased in universities across the country, the prevalence of cheating on programming assignments also increased [1]. Students are relying on others to pass their introductory programming courses rather than doing their own work. Plagiarism in introductory courses results in these students not having the foundation to successfully complete upper-level coursework. Researchers have been working on techniques to detect document plagiarism for many years [2,3]. The problem of detecting code plagiarism has been explored by computer scientists for more than thirty years. As of late, there have been several tools that help identify cases of plagiarism across programming assignments. Most of these products can be effective for complex programming assignments that result in large code bases and allow for a variety of problem-solving approaches. For introductory programming courses, concepts and language constructs are explored through small programs and simple algorithms, with less intrinsic variation among student solutions. It is not uncommon that existing tools will flag a high level of similarity between programs that were developed independently by students. Introductory programming assignments may generate answers at a level of structural similarity that makes identifying plagiarism much harder for lower-level courses.

Our research aims at exploring how and to what extent the student submissions to an online autograding platform can be used by the professor to identify probable cases of

plagiarism. At Texas A&M University, many instructors use Mimir Classroom [27] as the homework grading platform for the Introduction to Programming courses. Instructors prepare a test suite for each assignment, including function-specific test cases and tests that cover the whole program functionality. Students are told to first complete their homework assignments on their computer, testing their code locally with their own test cases and test datasets made available to students. They then upload their homework solution files to Mimir Classroom. Their submission is tested and graded immediately according to a set of test cases and criteria set by the instructor. The platform allows the student to see what their grade would be on that submission. They also have the ability to see detailed information about any of the visible test cases used for testing their code. Instructors can specify the maximum number of submissions for an assignment; for the courses we worked with in this research, assignments were configured to allow students to make as many submissions as they would like. They then can either edit their work and upload their new files or take the current grade. Mimir allows the instructor to download all assignment submissions for each student.

Besides managing assignment submission and automatic grading, Mimir also offers plagiarism support. Like most other tools identifying plagiarism, Mimir uses complex analysis of program structure, post-compilation analysis, and student user patterns to catch many of the tricks students employ when cheating. The Mimir platform offers three levels of detection. With current class analysis, the platform compares the solutions from current students; given the large number of students taking introductory courses, such analysis involves submissions from hundreds to thousands of students. With historical class analysis, submissions from previous semesters are also included. With web detection, Mimir compares student submissions with code

fragments available at repositories such as Stack Overflow [4] and RosettaCode [5]. Mimir also offers integration with one external plagiarism detection tool, as we discuss in Chapter X.

Although the plagiarism analysis provided by tools such as Mimir has been very useful, in this work we show that such information may lack the precision and granularity needed to help instructors to decide if it is appropriate to invest in further investigation. Our work explores how additional information about the solution development processes and student involvement in the course can be used to better support the instructor on assessing plagiarism cases.

From the bulk download of submitted code to an auto-grading platform, we can gather additional data about the student work. The work in this thesis created a framework that captures the data and makes it available for further analysis towards assessing how the extra information can help the instructor to better identify the code similarity cases to pursue further. Our framework captures every submission made to Mimir Classroom, including each individual file and the timestamp of each submission, integrates is with additional plagiarism detection information, and organizes the data for further analysis. All of this information can be used to reveal student code development habits that can be informative when reasoning on plagiarism. For example, submission with flagged code (i.e., with a large chunk of code detected as similar to code in another submission) from a student who has a submission profile very distinct from the rest of the class (e.g., very few submissions right before the deadline while the majority of students had many dozens of submissions) could be indicative of plagiarism. Another example is a student with an intermediate submission that replaced most of the functionality with new code that is similar to code from another student. The data available in our framework can also be analyzed to look at overall development patterns by students to identify students who possibly worked together.

With the inherent structural simplicity of the basic programs developed in introductory programming courses, code similarity analysis alone cannot identify plagiarism with perfect accuracy. Our goal is to aid instructors in identifying cases that may deserve more attention and investigation by integrating additional information about student work. It is not our intent to provide the professor with cases of plagiarism with a high level of certainty, but instead, give them insight into concerning behaviors.

2. RELATED WORK

2.1 Plagiarism Detection

Computer programming plagiarism detection methods have been around for decades. Ottenstein published a paper in 1976 discussing his method for detecting plagiarism in Fortran code. He quantified each assignment using a few different features and then quantitatively compared each submission to identify cases of plagiarism [7]. In 1987 Jankowitz published a paper evaluating methods for detecting plagiarism in Pascal programs [8]. This paper differed from its predecessors because it looked more at the order of procedures being executed, whereas other papers predating it just looked at features of the code, much like Ottenstein.

MOSS is a tool used for plagiarism detection of coding assignments. It works by comparing the similarity between programs [9]. This system is very effective for detecting student submissions derived from the same source through only small changes like changing variables names, adding whitespace, introduction or removal of comments, etc. MOSS tends to be better for higher-level programming courses, rather than lower-level, more introductory programming assignments [15]. Plagiarism is an increasing issue in introductory courses, where MOSS is not as accurate at identifying plagiarism cases.

MOSS is not the only widely used plagiarism detection software around, there are several others. Modiba et al. looked at multiple different plagiarism detection platforms: AC [10,11], CodeMatch [12], CPD [13], MOSS, and NED [14]. They then compared them to one another [15]. They found that each platform offered different advantages.

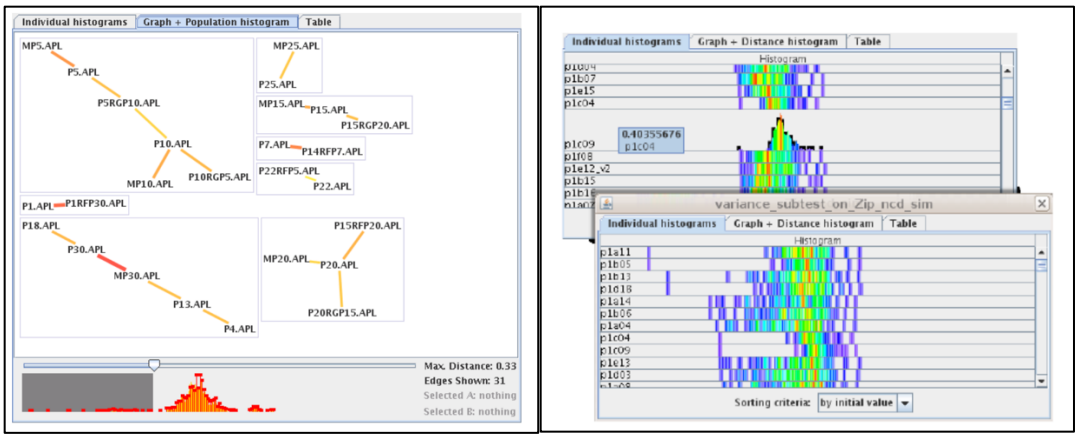


Figure 2-1 Examples of Visualizations Offered by AC. Reprinted from *Visualizing Program Similarity in the AC Plagiarism Detection System* by Manuel Freire [10].

AC seemed to be more advantageous when there is a very large number of submissions because it offers helpful visualization; whereas, CodeMatch is better for when you have only a small number of submissions because the number of results is the square of the number of submissions. Figure 2-1 shows a couple of examples of visualizations offered by AC. The left visualization is a plagiarism graph that shows parent and child accounts of plagiarism where the distance between each node is representative of similarity. The shorter the distance, the more similar the submissions are. The visualization on the right represents each submission and a histogram showing the similarity distance to each other submission.

CPD did not seem very advantageous when compared to the other systems because it only identified duplicated code, which is not very helpful since students often change the code just slightly to avoid detection. The authors concluded MOSS is a great tool for when programs are more complex. NED, at the time the authors published their paper, was only in the prototype stage, but seemed promising since it had the highest number of correctly identified cases of

plagiarism. Until it is in the production stage, and more readily available, it is not very advantageous.

Yan et al. designed TMOSS, an extension of MOSS [16]. MOSS is generally used to look at students' final submissions for an assignment, but TMOSS is designed to look at intermediate student work. The authors designed an Eclipse IDE that occasionally uploads the student work to the MOSS system throughout a programming assignment. The authors concluded that TMOSS can be used to identify excessive student collaboration and to help understand how students work from start to finish on a programming assignment. We aim to do something similar. We hope to capture student development patterns by capturing the intermediate code submissions to Mimir.

Tahaei and Noelle approach plagiarism detection in a way that does not involve detecting similarities of a student's submission to a variety of sources [17]. Instead, the authors use a feedback system that guides the student as they work, and the student is allowed multiple submissions of the assignment. The idea is that a student will improve on each submission if they follow the feedback given. The authors observe how the student progresses with each submission, attempting to identify behaviors that may be indicative of plagiarism. They compared their method to MOSS. They found that their method outperformed MOSS when identifying true plagiarism cases. The data and results proved that following the resubmission patterns of students could give insight into possible plagiarism cases. The main weakness of this method was the fact that it does nothing for students who make a single submission since it compares subsequent submissions.

One of the most concerning issues with plagiarism is the fact that it tends to affect student performance. Yan et al. confirmed through their study with the TMOSS system that students who tend to plagiarize and collaborate with others perform worse on exams [16]. Pierce and Zilles

conducted a study correlating plagiarism patterns and grades [18]. They found that there is a significant negative correlation between plagiarism and grades. The issue is, they could not identify whether weaker students tend to plagiarize or if plagiarizing contributed to the students performing worse than those who did not plagiarize.

Fonseca et al. looked into how detecting plagiarism early could be used to identify and address a student's difficulties [19]. In most cases, plagiarism is detected after the final submission of an assignment. The authors here decided to turn that process around and start detecting possible cases of plagiarism long before the final deadline. The idea was that if you could identify plagiarism early, you may identify where the student is struggling and address those issues. The authors' tool detected plagiarism in real-time and allowed the instructor of the course to view the data in real-time. They found that their tool did allow the professors to better understand where students were struggling in real-time due to identifying those who had plagiarized code. The professors could then intervene to help those students better understand the material. Although their tool seemed to perform well, they only tested with a fairly small sample space, so it cannot be determined how it would perform for a larger sample.

2.2. Autograding

It is no secret that introductory programming courses have high enrollment numbers [20, 21]. With these high enrollment numbers, it is much more difficult to grade programming assignments and provide students with meaningful and timely feedback. According to Chris Wilcox, autograding platforms are built to provide the student with meaningful feedback, automate the running, testing, and grading of the programs, and to help instructors create meaningful feedback [22]. Wilcox developed his own testing framework that consisted of a

back-end that performed the autograding and a front-end that allowed the instructor to incorporate real-time testing. They found that this framework performed well in most cases and saved a lot of time for the professor when compared to manual grading. Although their autograding system was successful, there were some issues that needed to be accounted for, such as non-terminating student programs, security, and performance to allow for an interactive system.

McBroom et al. investigated different techniques that could be used to identify how a student developed code over a semester in a junior CS course [23]. They focused on the autograding framework PASTA, which works by running a student submission against a set of professor defined test cases and then provides immediate feedback to the students. They looked at nine different features. For example, they looked at the percentage of attempts made three days or more before the deadline. They also looked at the percentage of tests passed on the first attempt. They found that students who started early and invested time into improving each attempt tended to have a final submission that was of much higher quality than their first submission. They also found that the students who tended to begin their assignments earlier were also the ones who tended to perform better on their final exam. The students who did not perform well on their final exam tended to be the students who did not submit their assignments at all.

Gramoli et al. mined submission data in 13 different subject areas over 3 years to demonstrate that autograding and feedback can be applied to computer science curriculum in general, not just programming courses [24]. They found that instant feedback and autograding frameworks are useful for subjects that are not assessing programming skills, in addition to those that are. They also found that the earlier a student starts to work on their assignment, the earlier they stop working on that assignment. They did find that students that start submitting earlier

tend to have better grades on those assignments, although there was not a significant correlation on performance. Overall, their main conclusion was that autograding and instant feedback is beneficial for students in courses other than those assessing programming skills.

Haldenman et al. worked to extend the information provided by two widely used, open-source autograding platforms, Web-CAT and Autolab, in order to provide more meaningful feedback for students [25]. Web-CAT and Autolab are both autograding platforms that allow you to have some set of test cases in order to test a student's code, and each test case has some hint associated with it [26]. After using Web-CAT in their introductory course, they found it to be hard to create hints that are not too vague or too detailed. They designed their methodology to analyze the results from an entire test suite rather than a single test case in order to guide the students in the right direction. After applying this methodology to two assignments in their introductory course, they concluded that the feedback provided to the students would be useful for correcting and understanding mistakes made on incorrect submissions.

Mimir Classroom is the autograding platform currently used for grading CSCE 121 labwork and homeworks. It allows the teaching staff to create assignments with a corresponding set of test cases. Students submit their assignments as many times as they want. On each submission they are able to see their score and what test cases they passed or failed. The teaching staff has the ability to show all test case information or limit how much information students are allowed to see about the test case. Mimir offers statistics on each assignment, like common errors across all submissions and the number of submissions per day. Mimir also offers plagiarism detection via MOSS and its own method.

Vocareum Classroom [28] is an autograding platform that was previously used for the grading of CSCE 121 labs and homeworks. It is similar to Mimir. It allows the teaching staff to

create assignments with a set of predetermined test cases. Students are able to submit as many times as they want. They are able to see their score each time they submit and which test cases were passed.

2.3. Student Fingerprint Identification

There have been efforts made in the past that incorporated information from student source code repositories (such as Github) to identify a software development fingerprint that captures the project development process. A fingerprint could be used to infer student behavior regarding code development and capturing how a student produces a final solution. Such fingerprint can also be used to help with identifying plagiarism as well [6].

The change history obtained from a source code repository captures a much richer view of program evolution than what can be captured by tracking the student submissions to an automatic-grading system like Mimir. Though, this much richer information would come with a price: most students in an Introduction to Programming course lack the experience to use source code control tools, such as GitHub. Requiring the use of such a tool – even with appropriate training – could add a hurdle to students who may already struggle with the material. An alternative approach would be to provide a working environment for the student that transparently captures the code as the student develops it.

We designed a virtual machine (VM) that pushed the student's work to their repository each time they compiled their program. The VM was meant to ease the difficulty of using GitHub for the students and allow us to easily access all of the GitHub information that revealed how the student evolved their solution. There are drawbacks to requiring students to use an instructor-provided VM as their development environment. In this work, we explore how the

limited development history information available from a submission system can aim at assessing how the information from Mimir Classroom submissions can be used to further refine the plagiarism information it provides.

3. THE PRAISE FRAMEWORK

We have created the PRAISE (**PR**ogramming **A**ctivity **I**ndicators of **S**tudent **E**ffort) framework that pulls student activity information together from several sources, making them available for data analysis during a course and across semesters. We designed PRAISE with three goals. First, PRAISE intends to provide instructors with a dashboard for use during the semester. This dashboard enables the instructor to observe overall student behavior on programming assignments, possibly identifying unusual difficulties with an assignment or struggle scenarios particular to some students. Second, PRAISE can aid instructors to detect plagiarism by enhancing the existing code similarity information with indicators of the student engagement in course assignments. Third, PRAISE aims at building a repository of student activities useful for mining the relationship between students' track record in course tasks and their overall performance in the course.

Regarding the plagiarism detection support, PRAISE was designed to build on top of the analysis provided by Mimir and MOSS. MOSS, discussed in Section 2.1, is a widely used plagiarism tool. Mimir offers plagiarism analysis in two forms: they can query the MOSS system (using the instructor's authentication identification for that system) and they can report on their own analysis. Mimir, described in Section 1, provides general analytics about assignments like overall grades, grades per test case, and the number of submissions per day. PRAISE goes beyond overall startistics, aiming to add more insight into overall course trends and the programming fingerprint of each student by analyzing individual submissions from students and contrasting them with the overall patterns in the course.

We bring together data from submissions on homework assignments and labs, attendance, exam grades, and Piazza participation. The data from the sources do not tell much of a story when looked at individually, but when this data is harnessed in combination, overall trends about the course and the coding behavior of individual students may start to emerge. We designed PRAISE as a data integration and repository tool to enable data analytics on fine-grained activity information in programming courses. We support data collected from multiple offerings, which allows for the comparison of data across semesters and instructors, even when the offerings adopt different homework or lab assignments, leading to a deeper analysis and understanding of the course.

The first prototype for PRAISE is tightly coupled with Mimir. For any assignments that are submitted using Mimir, we are able to access the source code for each individual submission from a student. PRAISE processes these submissions to get their timestamps, the percentage of lines changed from one submission to the next, and the total number of submissions per student. Our framework offers a set of visualizations using the data gathered. PRAISE also enhances the traditional plagiarism ranking by leveraging information gathered from a student's submission pattern.

3.1 Analytics Provided by Mimir

Figures 3-1 and 3-2 depict the data analytics currently provided by Mimir Classroom, showing the data for the first programming assignment in the Fall 2019 CSCE 121 offering. As we describe in this chapter, PRAISE's data analytics capabilities go much further than what the Mimir Classroom product currently offers. Another disadvantage of Mimir's current support for data analytics is that, in our experience, it fails to show any analytics for large datasets. Even

after dozens of attempts spread across different days, we failed to retrieve to obtain any results for most of the homeworks in the course.

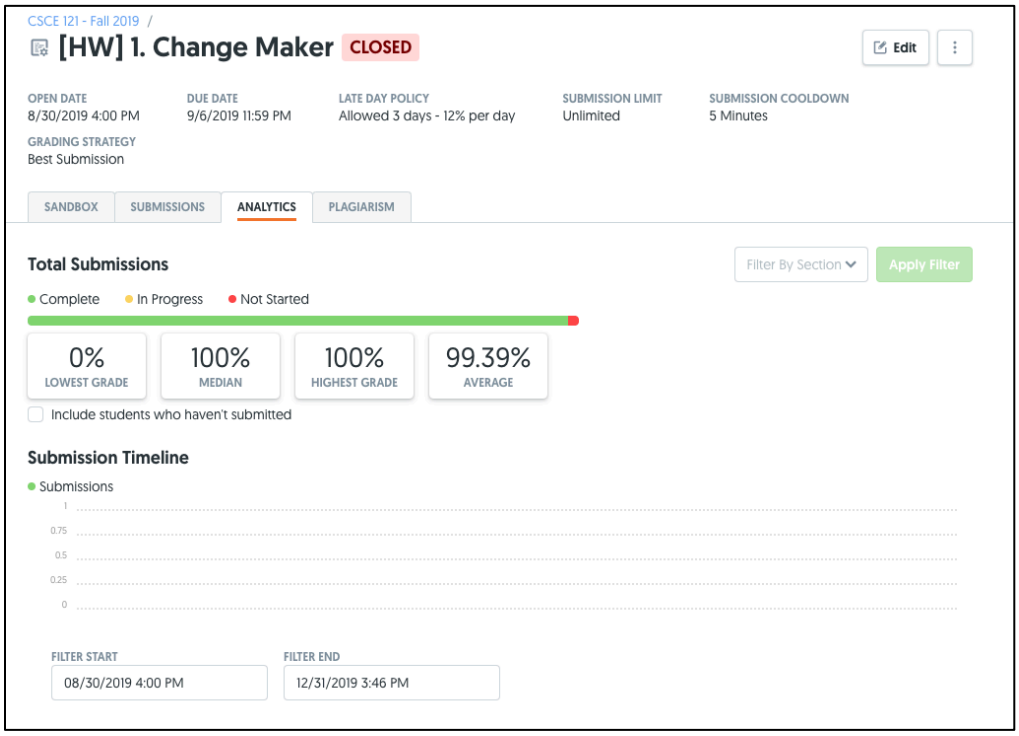
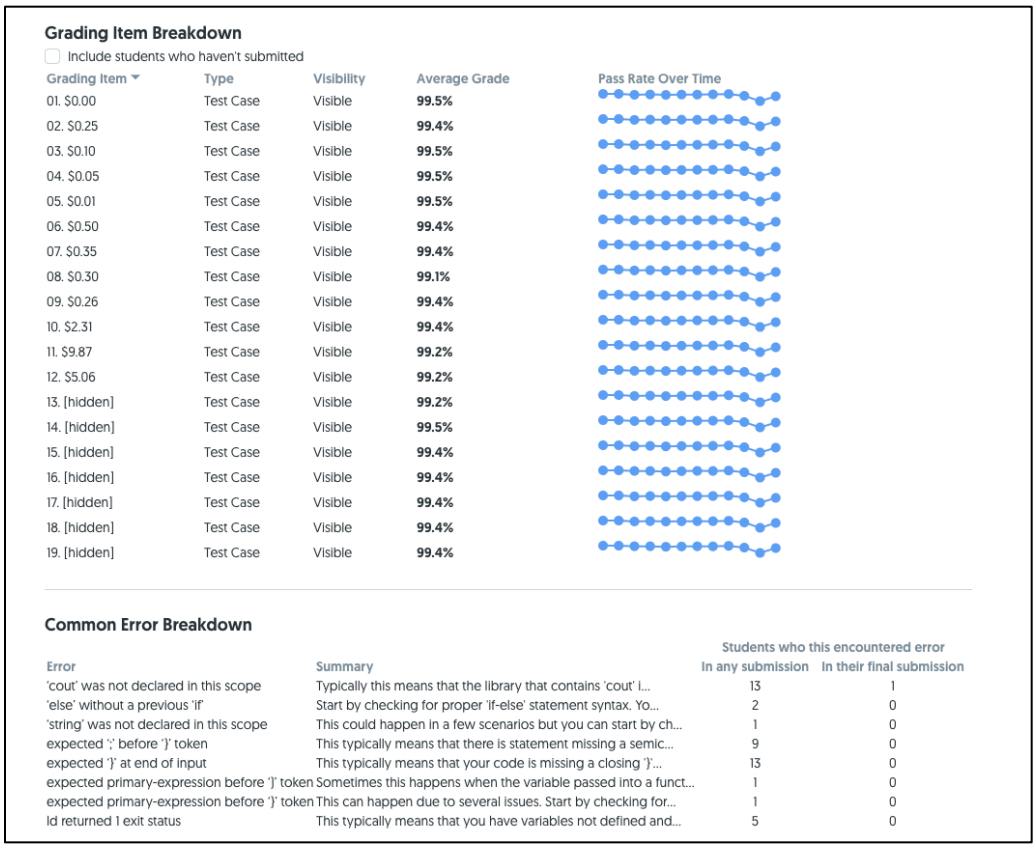


Figure 3-1 Snapshot of the Analytics from Mimir Classroom (Part 1)



Grading Item Breakdown

Include students who haven't submitted

Grading Item	Type	Visibility	Average Grade	Pass Rate Over Time
01. $0.00	Test Case	Visible	**99.5%**	
02. $0.25	Test Case	Visible	**99.4%**	
03. $0.10	Test Case	Visible	**99.5%**	
04. $0.05	Test Case	Visible	**99.5%**	
05. $0.01	Test Case	Visible	**99.5%**	
06. $0.50	Test Case	Visible	**99.4%**	
07. $0.35	Test Case	Visible	**99.4%**	
08. $0.30	Test Case	Visible	**99.1%**	
09. $0.26	Test Case	Visible	**99.4%**	
10. $2.31	Test Case	Visible	**99.4%**	
11. $9.87	Test Case	Visible	**99.2%**	
12. $5.06	Test Case	Visible	**99.2%**	
13. [hidden]	Test Case	Visible	**99.2%**	
14. [hidden]	Test Case	Visible	**99.5%**	
15. [hidden]	Test Case	Visible	**99.4%**	
16. [hidden]	Test Case	Visible	**99.4%**	
17. [hidden]	Test Case	Visible	**99.4%**	
18. [hidden]	Test Case	Visible	**99.4%**	
19. [hidden]	Test Case	Visible	**99.4%**	

Common Error Breakdown

Error	Summary	Students who this encountered error: In any submission	Students who this encountered error: In their final submission
'cout' was not declared in this scope	Typically this means that the library that contains 'cout' i...	13	1
'else' without a previous 'if'	Start by checking for proper 'if-else' statement syntax. Yo...	2	0
'string' was not declared in this scope	This could happen in a few scenarios but you can start by ch...	1	0
expected ';' before '}' token	This typically means that there is statement missing a semic...	9	0
expected '}' at end of input	This typically means that your code is missing a closing '}'...	13	0
expected primary-expression before '}' token	Sometimes this happens when the variable passed into a funct...	1	0
expected primary-expression before '}' token	This can happen due to several issues. Start by checking for...	1	0
ld returned 1 exit status	This typically means that you have variables not defined and...	5	0

Figure 3-2 Snapshot of the Analytics from Mimir Classroom (Part 2)

3.2 Data Integration

The PRAISE framework is organized in terms of course offerings. Instructors or specify a course offering through an open-standard file specification format such as JSON. The specification lists the student activities to be integrated into the PRAISE repository, such as programming assignments, exams, class attendance, or Piazza participation statistics. Figure 3-3 displays the framework's UML diagram, which gives insight into all of the information we are storing. For brevity, in this section we focus on the description of the PRAISE framework components related to programming assignments.

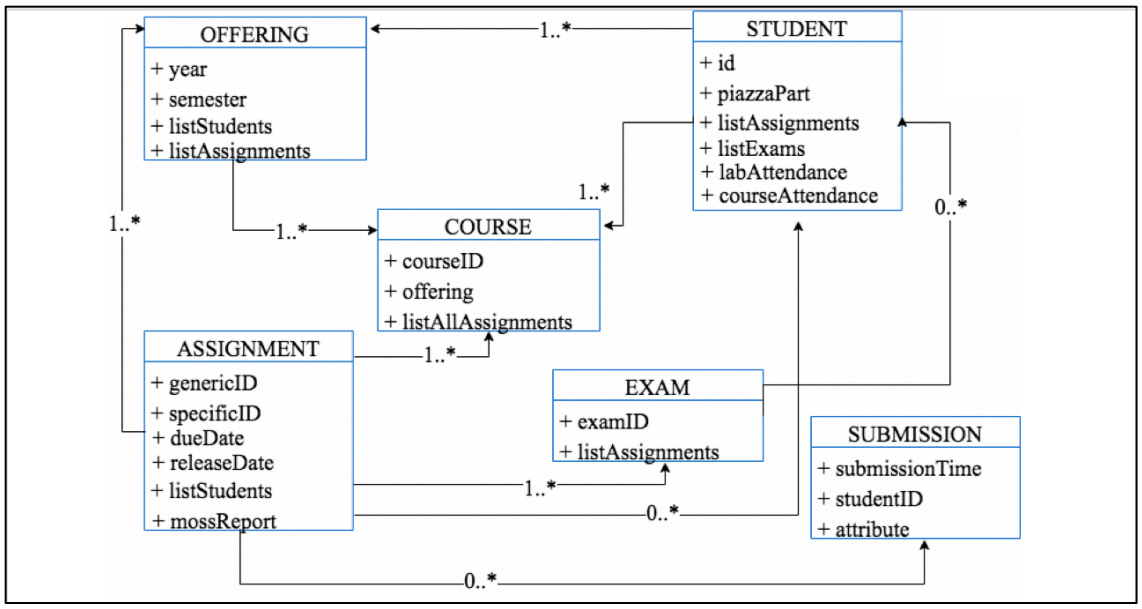


Figure 3-3 PRAISE UML class diagram

In order to capture the overall trends and individual coding fingerprint for a student, we must first have access to the information from Mimir. Mimir allows the instructor to download a zip file containing every submission from a student for a specific coding assignment. The zip file contains a directory for each student. The student directory then has a directory for each submission the student has made along with the source code for that submission. This zip file has to be processed so that all the necessary information can be extracted and then used. Figure 3-4 depicts the PRAISE design.

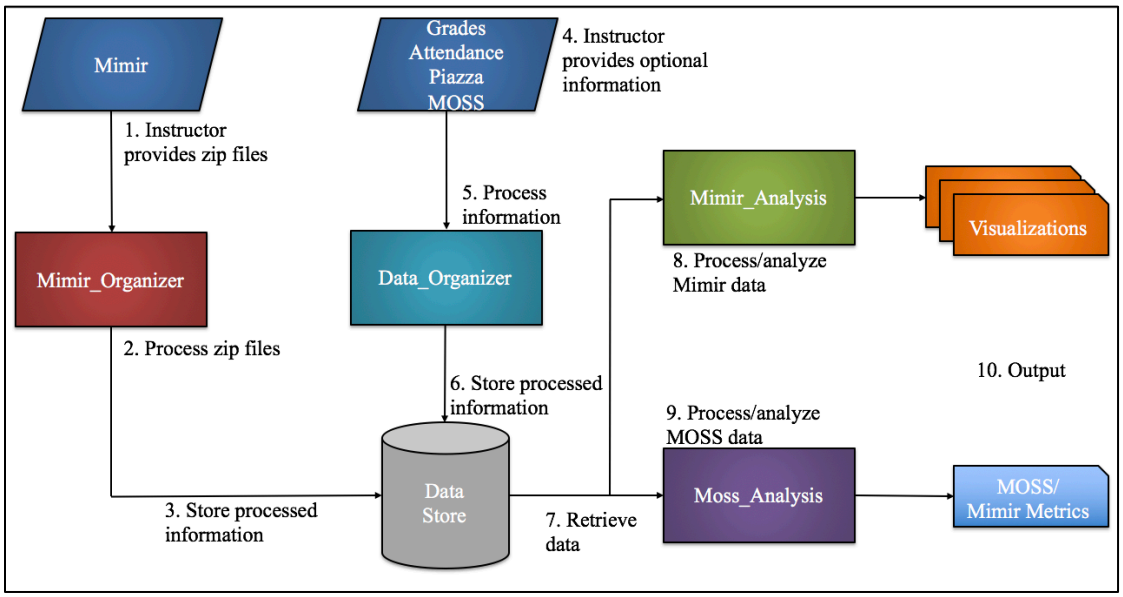


Figure 3-4 Visual depiction of the layout and purpose of the components in the PRAISE framework

The component Mimir_Organizer, a script, handles extracting the files from the zip file. An instructor can download the zip file from Mimir. The zip file will need to be stored in a directory. The instructor provides Mimir_Organizer with the path to the directory containing the zip file. The current prototype also requires the output path where the instructors would like the extracted data to be saved, but the next version of PRAISE will take care of the persistent storage of anonymized submissions. If there are multiple zip files (i.e., for multiple assignments) saved in the directory, the script will process all of the zip files at once. The script extracts everything from the zip file, anonymizes the directories that were previously named using student emails for cross-semester analysis, and restores all original timestamps to the files. This is the first step in preprocessing the information.

The second step prepares the information to be stored in the data store. The script loops through each student submission and extracts the total number of submissions, the timestamp for

each submission, the number of lines changed for that submission compared to the last, and the percentage of lines changed between that submission and the last. In out first PRAISE prototype, the data store is a csv (comma-separated values) file, but a future version of PRAISE will need to deploy a database to support the target scalability of the system to hundreds of courses offerings, thousands of students and potentially millions of programming assignment submissions. These csv files, and later the database, contain all of the assignment information, meaning that anyone wanting to perform their own data analysis has the capability of using the information from the data store as they wish.

The PRAISE framework is meant to be flexible. It can process optional information upon request from the user. As mentioned earlier, there is a great deal of data that is available for the CSCE 121 course used to exercise the PRAISE prototype. An instructor has the option to provide additional information from attendance records, exam grades from Gradescope, Piazza, and assignment grades. When this information is provided, the Data_Organizer is responsible for processing the data from its raw format to the format expected by the tools (e.g., python scripts) to be used to analyze the data. Again, data from these sources is completely optional. When they are provided, other scripts in the PRAISE framework can provide the analysis associated with them. If they are not provided, the student submission information from Mimir will be the only information analyzed.

Mimir_Analysis is a Jupyter Notebook [29] that can be harnessed in order to process and visualize all wanted information. It incorporates a menu that asks the user a few questions in order to identify where the pertinent csv files are located, the assignments they want data analysis on, and the due dates for those assignments. Once this information is provided, the tool generates a series of graphs based on what the user requested. If the user opted to only analyze

the Mimir submission information, eight different graphs are generated, as illustrated in the figures in Section 4. The additional data the user chooses to include determines what other graphs will be generated. All graphs will be displayed immediately in the Jupyter Notebook and saved as a single pdf file for the user to download and view as needed.

3.3 Plagiarism Enhancement

Student 1	Student 2	File	Student 1 Similarity	Student 2 Similarity	Lines Matched
f02ee921a3ecb8680 4e2d2733062b275	823d36aefee6c20ac 8ab7a712ac142af	functions	99	99	101
86ba4f8f35a594500 a978f19bb3f75d5	f02ee921a3ecb8680 4e2d2733062b275	functions	99	99	101
ff1068bef0f49798e 7d19289309395f4	5e7c192ce9f205cff cf32c7babc4ee45	functions	99	99	155
ff1068bef0f49798e 7d19289309395f4	53354af1f4baa8b6e 5cca73ac1bb441d	functions	99	99	101
ff1068bef0f49798e 7d19289309395f4	823d36aefee6c20ac 8ab7a712ac142af	functions	99	99	101
51b848140ecfe885 70f4c7a508694ca2	86ba4f8f35a594500 a978f19bb3f75d5	functions	99	99	107

Table 3-1 MOSS report for a homework assignment (shown with anonymized student information)

Mimir offers a plagiarism report from MOSS, but we have discovered that the MOSS report offered by Mimir does not exclude starter code, which skews the results. To have better control of the MOSS configuration used on the plagiarism detection, we obtain MOSS reports directly from its server. We have written a python script that directly runs the MOSS service with the homework assignment submissions downloaded from Mimir. The MOSS report is a link to a webpage, so we have also written a python script to extract and convert the necessary information from the webpage to a csv file for easy analysis. Table 3-1 shows the MOSS report for the homework assignment Tweets.

From Table 3-1, we can see that MOSS reports plagiarism by listing a pair of students, a file, a percentage of similarity for each student's code, and the number of lines matched. It is important that MOSS is able to analyze code at a structural level, so it can identify similarities even when variable names are changed, whitespace has changed, or comments have changed. As discussed in Section 1, plagiarism cannot be determined strictly from a MOSS report, so instructors often have to analyze more data in order to conclusively determine if a submission is plagiarized. Instructors will look at a variety of features in order to conclusively determine plagiarism, usually through inspection of coding intricacies such as typos in comments, commenting style, unusual placement of braces and brackets, peculiar algorithm choices, variable name choices, operand order in long expressions, and whitespace placement. With the increasing adoption of code editors and Integrated Development Environment (IDE) tools that take care of code formatting for the student, many of the idiosyncrasies that are useful to pinpoint plagiarism disappear as the tool adjusts element placement to conform to best practices. With our work, we aim to enhance the traditional approach to plagiarism identification by enhancing the MOSS report with features that capture additional information about the software

development process. By making available information for each student such as the number of submissions, their timestamps, and their rate of change between submissions, we expose to the instructor the coding fingerprint for each student in their efforts to submit their final solution. Our experience, as reported in Section 4, shows that students tend to submit many solutions as they receive instant feedback from the autograding system. Such submission patterns provide more insight into the probability of plagiarism.

This work explores a set of features to assess their effectiveness in reflecting student submission patterns that are likely to be linked to plagiarism. We decided to focus on the total number of submissions, the time between a student's first submission and their[1] last submission, and the percentage of lines changed for the last submission. In addition to these features, we also consider how these measures for a given student compare to the overall set of students in the course offering: we provide the raw metrics and their percentile placement in the student group. PRAISE is intended to be a platform to enable experimentation with additional features.

Moss_Analysis is another Jupyter notebook that allows for customization through user-inputted information like the path to the MOSS csv file and the path to the Mimir Analysis file. We take the MOSS report and for any two students listed in the MOSS file, we use the Mimir data to gain further insight into their programming behaviors. Moss_Analysis creates a new file that contains the MOSS information in addition to the Mimir metric and the respective percentiles for those metrics for each student.

[1] In this document, we use the "their" for singular instead of his/her.

4. EXPERIMENTATION

This chapter reports in our experience using the PRAISE framework to assist on plagiarism detection.

4.1 Data Sets

We have collected data from the Fall 2018 and Fall 2019 course offerings for CSCE 121: Introduction to Program Design and Concepts at Texas A&M University. From the Fall 2018 course, we have 10 different assignments with at least with at least 300 students doing each assignment. For the Fall 2019 offering, we have a total of 10 homework assignments with 595 or more students attempting each assignment. Generally, there are more students completing the assignments that occur early in the semester compared to those that occur later as many students may choose to drop the course. For the purposes of this research, all identifying student information has been anonymized.

4.2 Usage Scenarios

We demonstrate our experience with the PRAISE framework by showing the analysis provided by PRAISE for the homework assignments from Fall 2019. When illustrating the analysis of a single assignment, we chose two of the assignments: the first (Change Maker) and the eighth (Tweets) homework assignments of the course to capture codebases at different levels of complexities. Change Maker was completed by 657 students through a total of 1,831 submissions. Tweets was completed by 593 students with a total of 9,408 submissions. The specification for both assignments is included in Appendix A.

4.2.1 Data Visualization of Mimir Submission Patterns

PRAISE shows several interesting features captured from the Mimir submission analysis. First, we show some of the information that PRAISE can provide for assignments across the semester. Figure 4-1 depicts the distribution for total submissions per day. Figure 4-2 shows the distribution of first submissions for each day and Figuer 4-3 shows the numbers of last (final) submissions per day across all assignments.

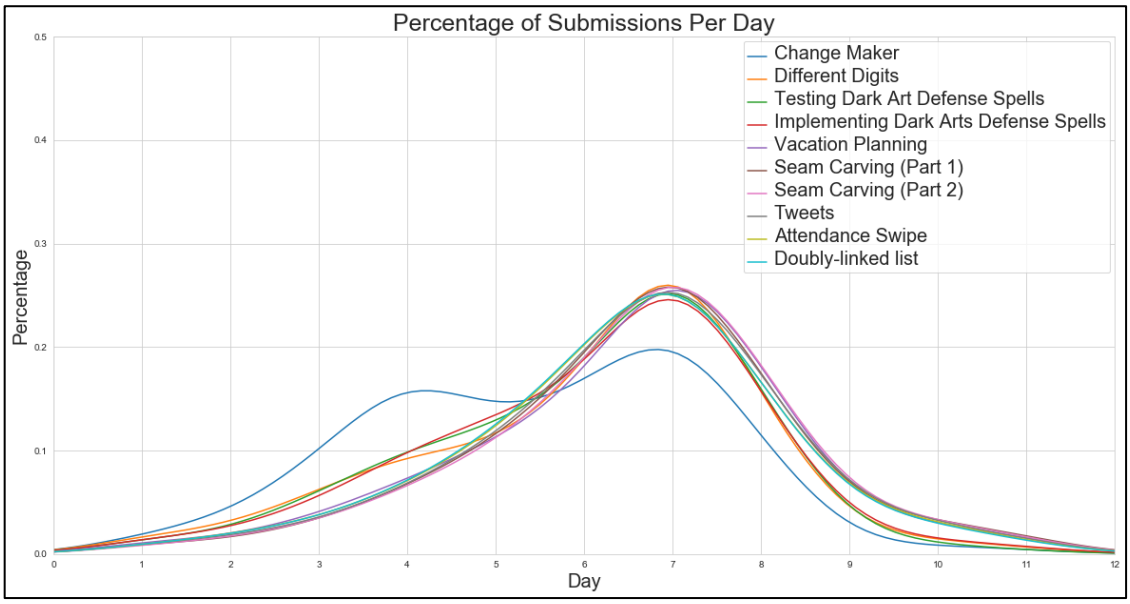


Figure 4-1 The Percentage of Submissions Per Day for All Assignments

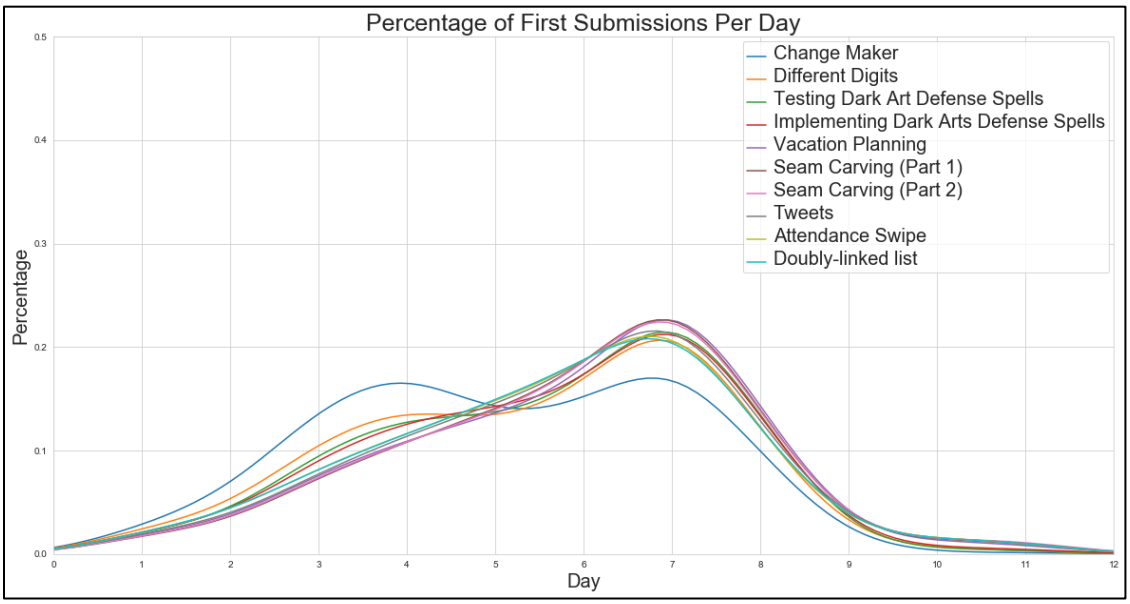


Figure 4-2 The Percentage of First Submissions Per Day for All Assignments

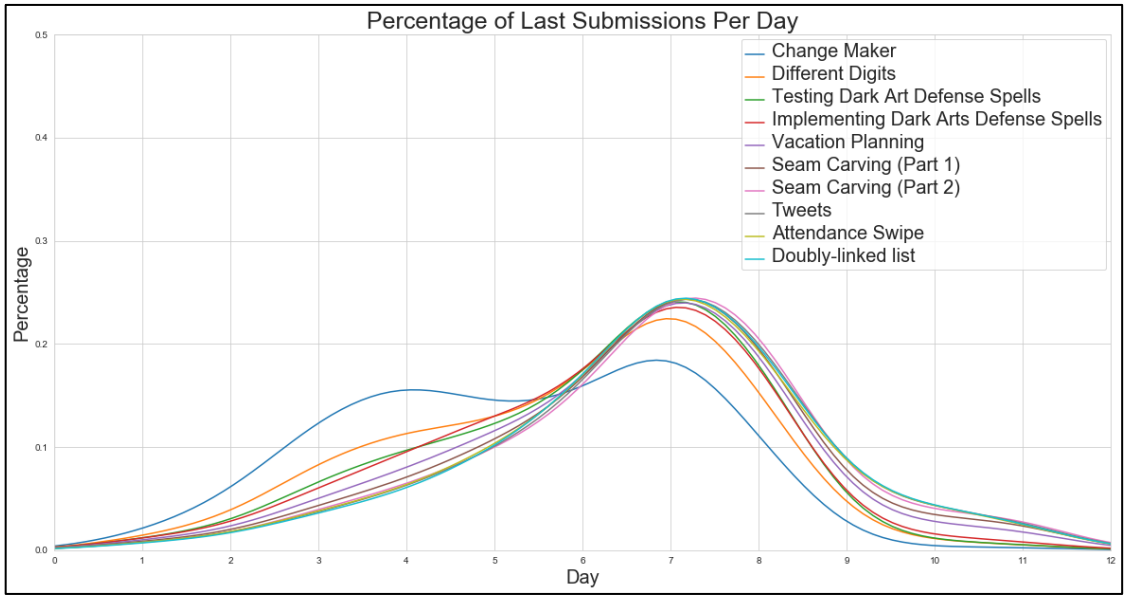


Figure 4-3 The Percentage of Last Submissions Per Day for All Assignments

Next, we illustrate the information PRAISE provides for specific assignments. We look at the total number of submissions per day for a single assignment; Figure 4-4 shows the distribution of the total number of submissions made for Change Maker, the first homework assignment from the Fall 2019 offering. We can see that as we approach the due date, September 6th at 11:59 pm, the number of submissions per day generally increased, as expected. September 5th had the greatest number of submissions, with about 700 total. Also, the last 3 bars represent late submissions for the assignment. This assignment had very few late submissions. This can most likely be attributed to it being the first assignment of the course and its relative simplicity.

Figure 4-5 shows the distribution of the total number of submissions per day for the homework assignment Tweets, which was the eighth assignment of the semester, so it was relatively more difficult than Change Maker. With this assignment, we have an almost normal distribution, with the total number of submissions peaking on October 31st, two days before the due date, with about 4800 submissions. Comparing this to the 700 max submissions for the Change Maker assignment, we notice that with more difficult assignments, there may be a drastic increase in the number of submissions made. Looking at the late submissions for this assignment, we can see that there were many more late submissions made compared to the Change Maker assignment. This makes sense considering the increased difficulty from one assignment to the next.

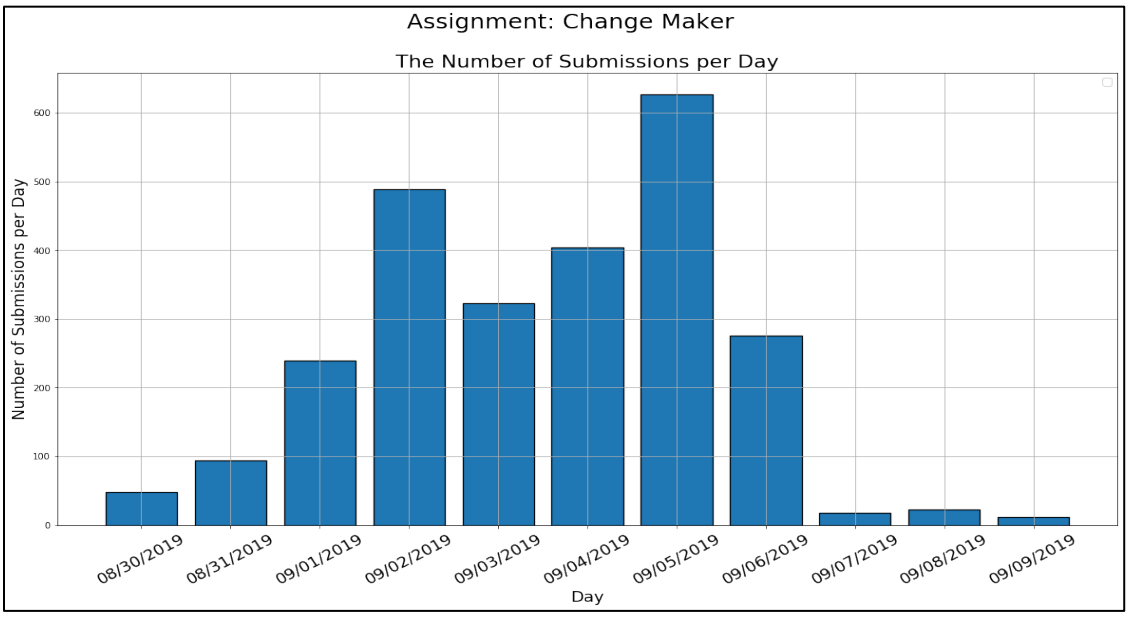


Figure 4-4 The distribution showing the total number of submissions made per day for the assignment Change Maker from Fall 2019.

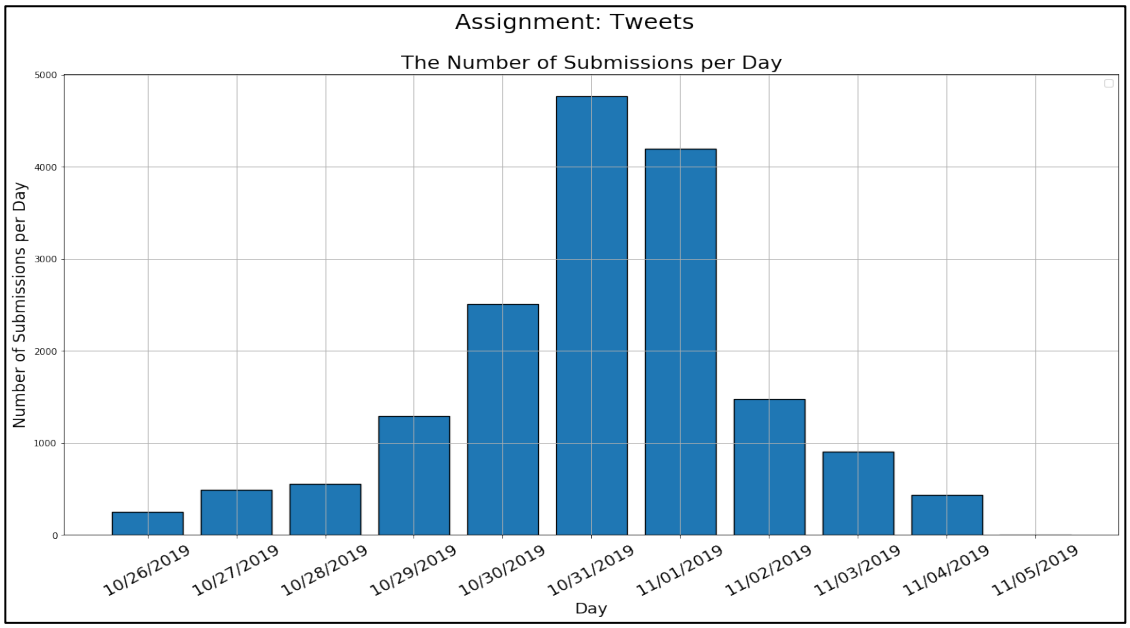


Figure 4-5 The distribution showing the total number of submissions made per day for the assignment Tweets from Fall 2019.

Not only does PRAISE displays the total number of submissions per day, but we also visualize the number of first and last submissions made per day. These distributions give insight into questions like: “How many students are starting the assignment early?”, “How many students are procrastinating?”, and “How many students are finishing the assignment relatively early?”. These are all questions that can be explored by looking at these two distributions. For example, Figure 4-6 and Figure 4-7 show the distribution of the first and last submissions made per day, respectively, for the Change Maker assignment. These graphs make it very easy to see that there were very few students who started early and finished early. Surprisingly, there were a few students who did not even start the assignment until after the due date. Figure 4-8 and Figure 4-9 show the same distributions for the Tweets homework assignment. We see some similar trends. There were very few students who started early and finished early. There were even more students who made their first submission after the due date. These are interesting observations that could be explored further through additional analysis, like correlating the time of the first and last submission with the final grade received on the assignment.

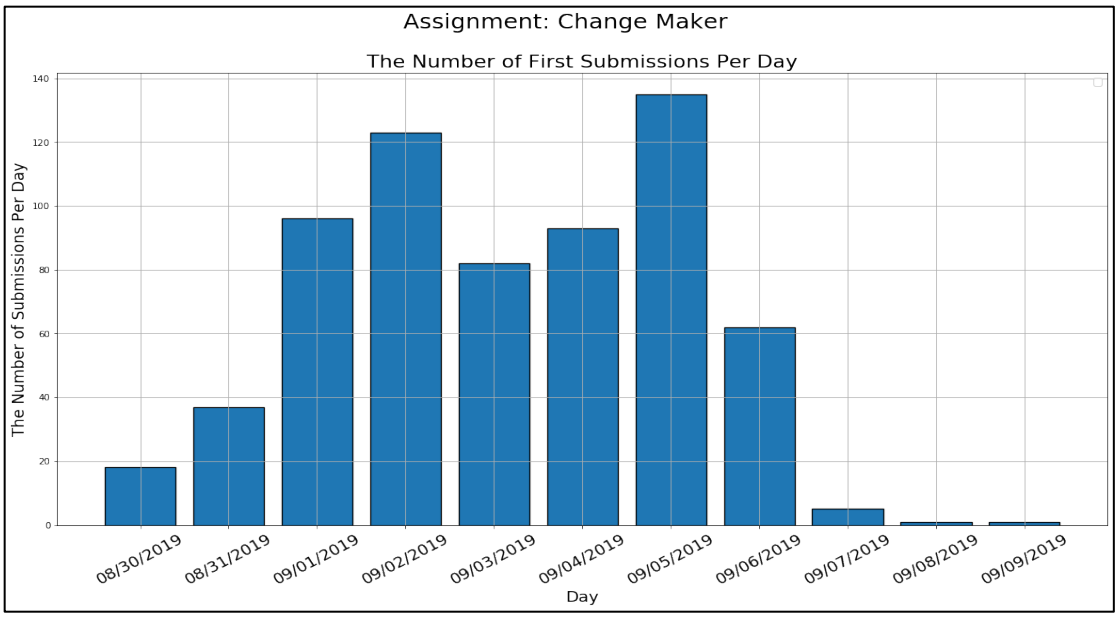


Figure 4-6 The distribution showing the total number of first submissions made per day for the assignment Change Maker.

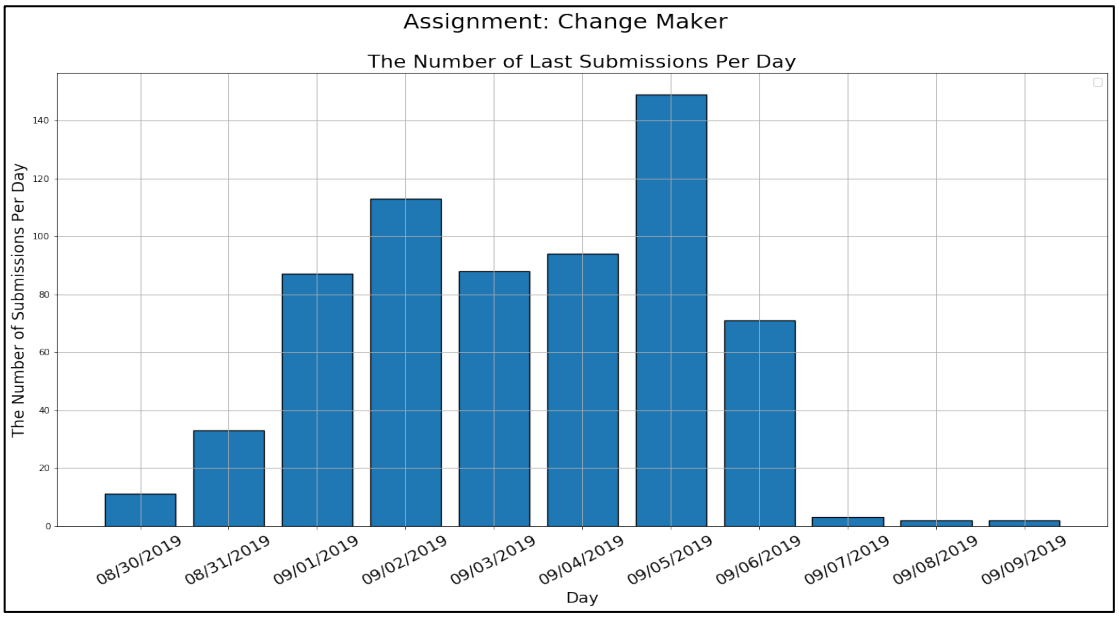


Figure 4-7 The distribution showing the total number of last submissions made per day for the assignment Change Maker.

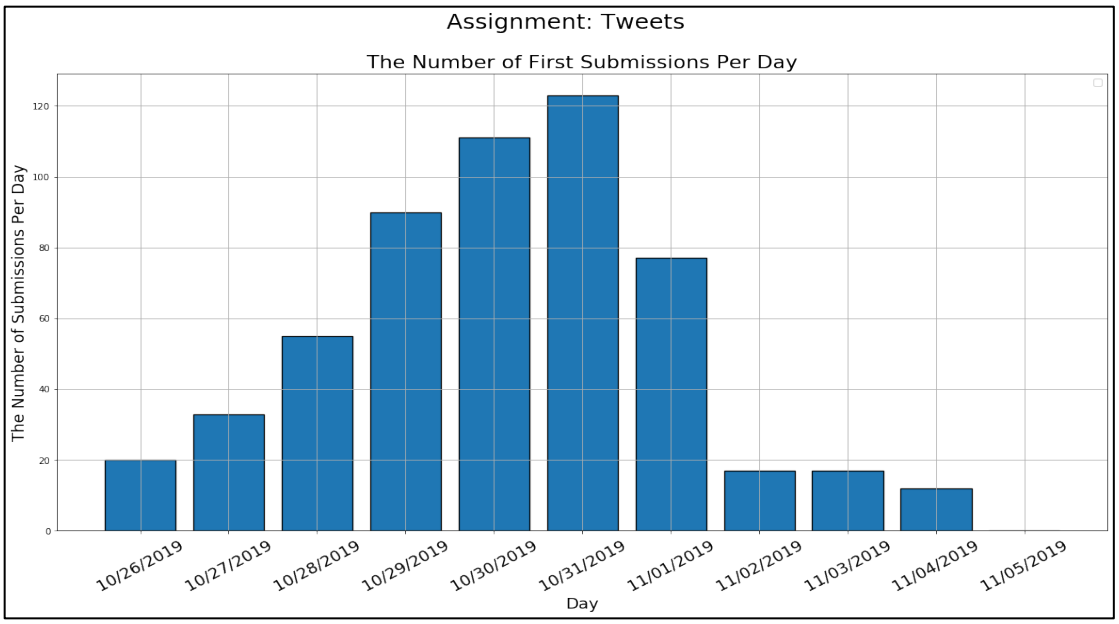


Figure 4-8 The distribution showing the total number of first submissions made per day for the assignment Tweets.

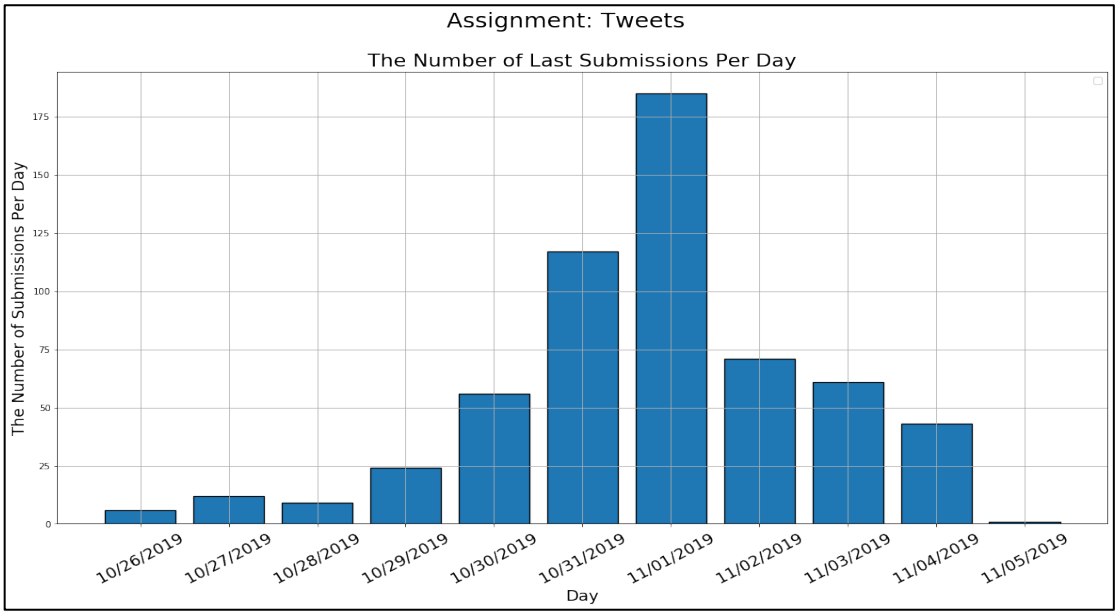


Figure 4-9 The distribution showing the total number of last submissions made per day for the assignment Tweets.

In addition to the standalone distributions mentioned previously, PRAISE can combine the information into a single graph for ease of observation. Figure 4-10 and Figure 4-11 show the combined distributions for the Change Maker and Tweets assignments, respectively. Instead of displaying the counts for each kind of submission per day like with the other graphs, we used percentages here instead. These graphs are meant to easily combine the previously mentioned distributions into a single graph for easy observation and understanding. When looking at the two figures, we can see that there were many more submissions per day for the Tweets assignment compared to Change Maker.

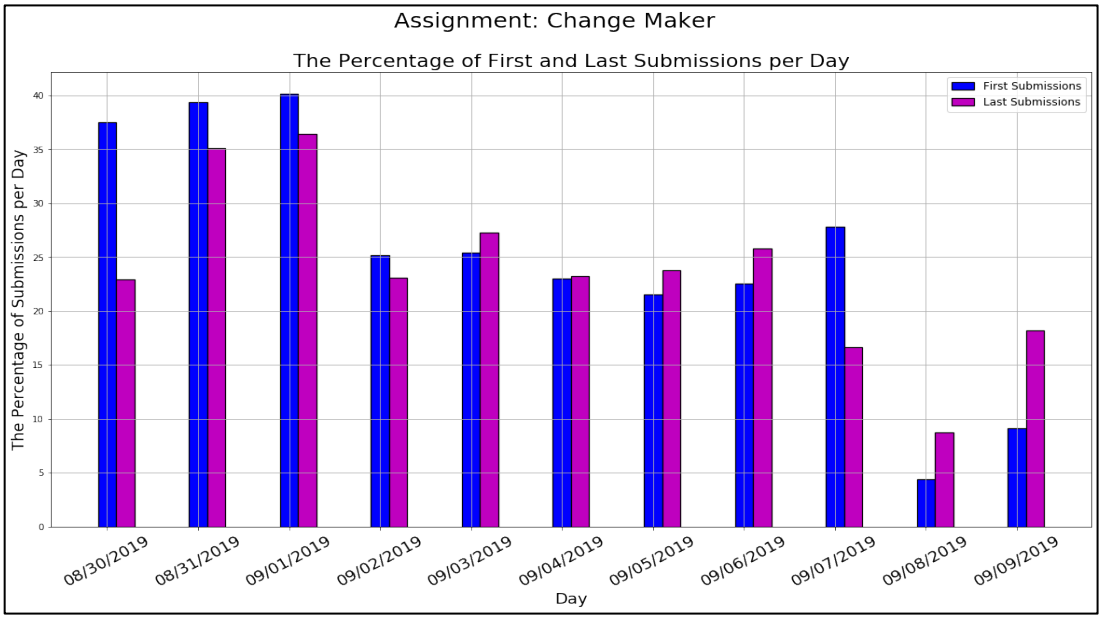


Figure 4-10 The percentage of the first and last submissions made per day for the Change Maker assignment.

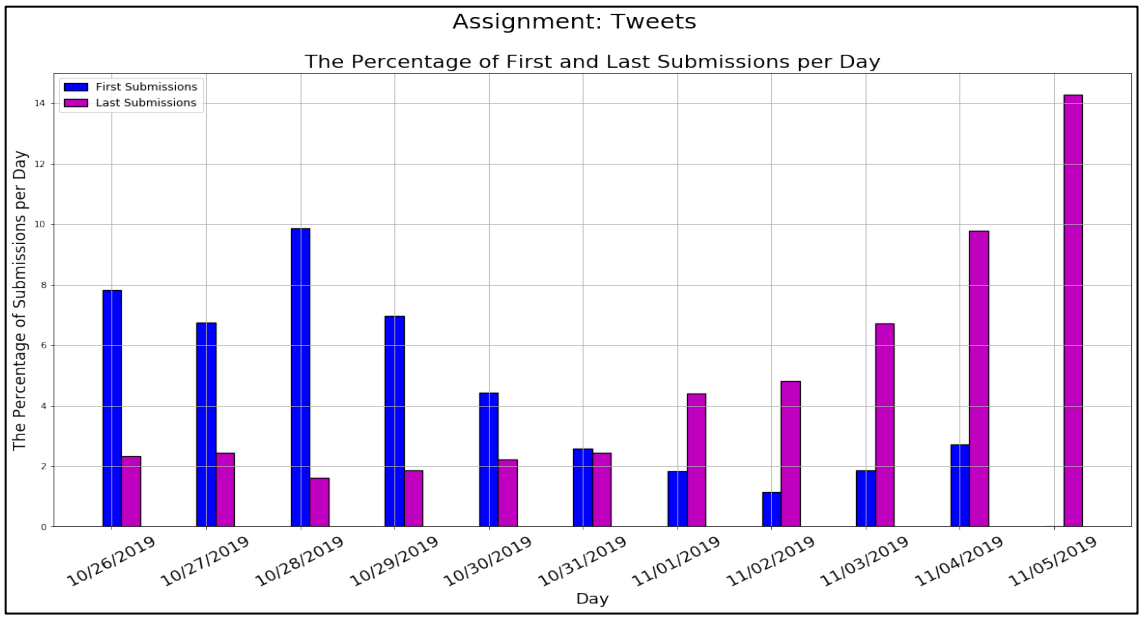


Figure 4-11 The percentage of the first and last submissions made per day for the Tweets assignment.

In addition to the distribution graphs, PRAISE offers several other visualizations based on the data analysis of the Mimir submissions. Figure 4-12 displays the total number of submissions for a student versus the time of their last submission for the Change Maker assignment. The motivation for exploring this graph was the plagiarism detection portion of the framework. PRAISE is able to display the notion of "programming fingerprint" of each student. The idea here is that students who have a very late last submission time, with very few submissions (in relation to the overall behavior in their group) may have an unusual pattern of activities leading to the completion of their homework. With the intent of further capturing student coding fingerprints, we display a similar graph with the time of the first submission rather than the time of the last submission, which is shown in Figure 4-13. For these graphs,

PRAISE also provides markers for the assignment release date, the assignment due date, the average number of total submissions, and the average submission time.

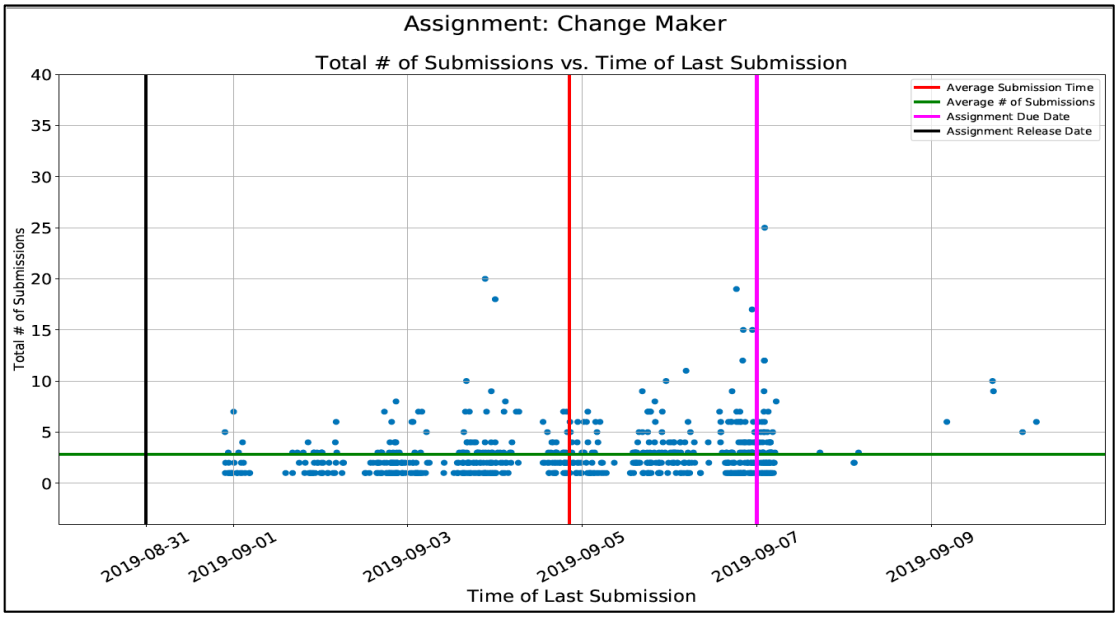


Figure 4-12 Graph of the total number of submissions versus the time of the last submission for each student for the Change Maker assignment.

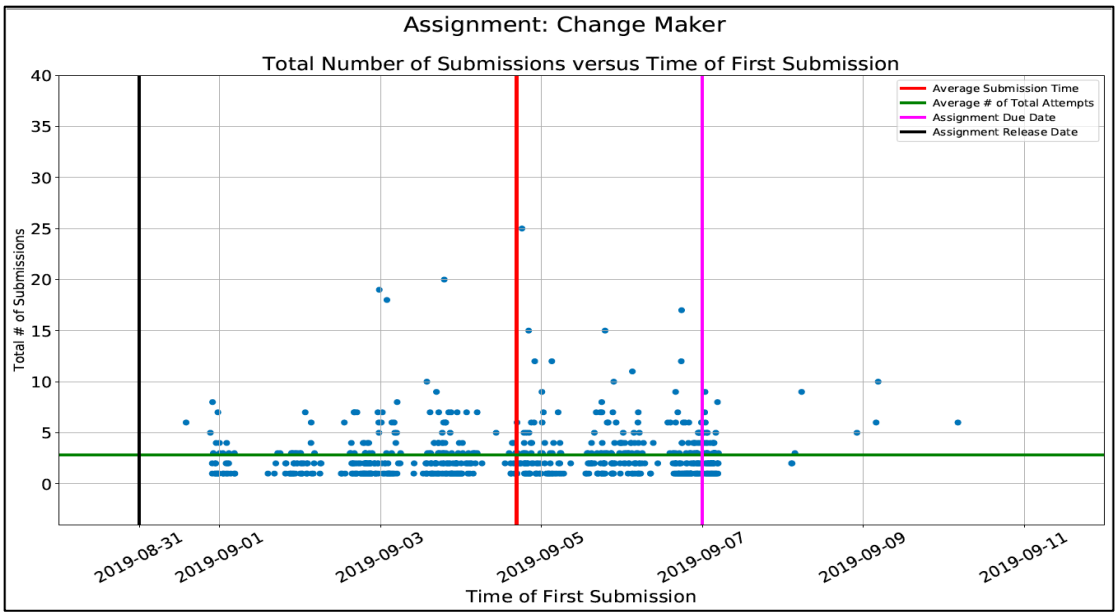


Figure 4-13 Graph of the total number of submissions versus the time of the first submission for each student for the Change Maker assignment.

In addition to all of the other graphs mentioned previously, PRAISE automatically provides two additional graphs to the user. Figure 4-14 and Figure 4-15 show the percentage of lines changed for the last submission and the percentage of lines changed versus the total number of attempts, respectively. We decided to include both of these graphs in PRAISE because the percentage of lines changed for the last submission may be of interest when trying to understand a student's coding behavior. Looking only at the percentage of lines changed for the last submission can be misleading, especially for the less complex assignments with a small number of lines of code. There are plagiarism scenarios that involve a large percentage of lines changed in the final submission, for example when a student gives up on fixing the problems in their code and submits new code obtained through collaboration/interaction with other students or tutoring services. There are instances where a student only submits once or twice, so the percentage of lines changed for their last submission is high, but this does not mean that they used other

sources to complete their homework. Looking at Figure 4-14 and Figure 4-15, we can see that the majority of those who had a high percentage of lines changed for their last submission had a very low number of total submissions. Of course, those who submitted right around the due date, only had a few attempts and had a large percentage of lines changed for their last submission while exhibiting high code similarity level – as reported by tools such as MOSS – would be a more likely case of plagiarism; and therefore, would need to be investigated further.

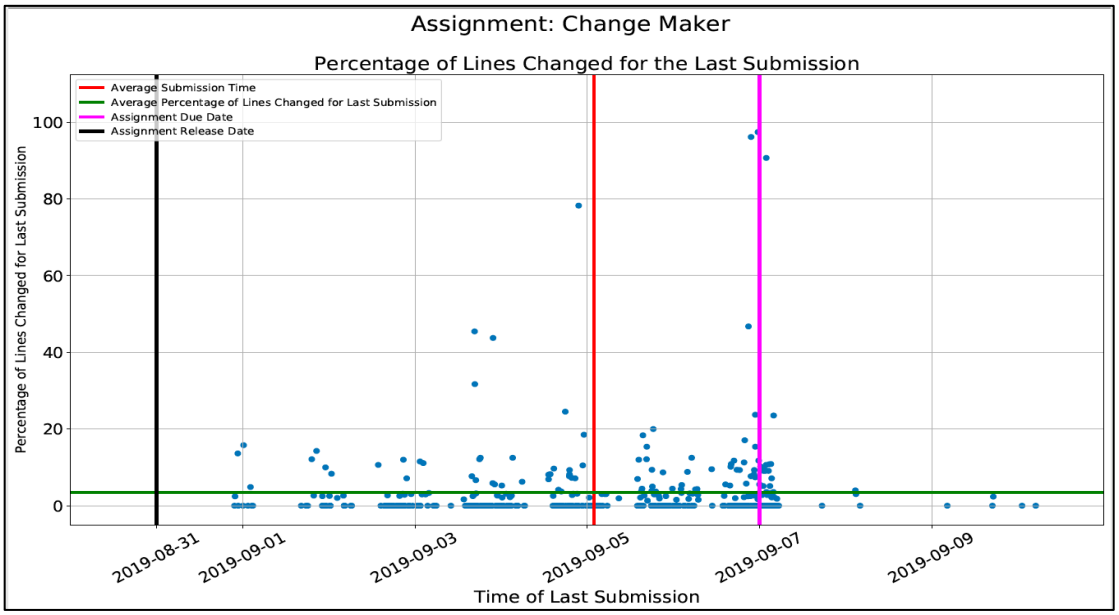


Figure 4-14 Graph displays the percentage of lines changed for a student's last submission for the Change Maker assignment.

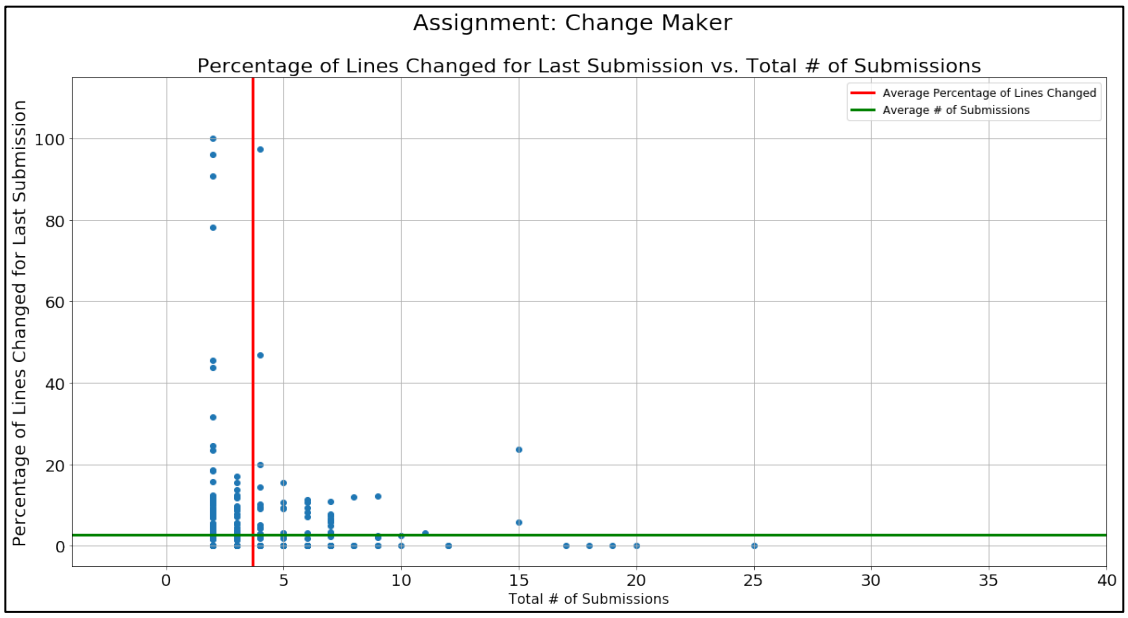


Figure 4-15 Graph displays the percentage of lines changed for a student's last submission versus the number of submissions made by that student for the Change Maker assignment.

4.2.2 Additional Data Visualizations

In addition to the default graphs mentioned above, PRAISE supports the production of several optional graphs. For example, Figure 4-16 and Figure 4-17 harness the assignment grade information provided by the user in order to show final assignment grade versus time of the first and the last submissions, respectively. These graphs may give insight into overall trends related to grades outcome. For example, does the data show that those who start working earlier get higher grades than those who wait to begin their work? Or does the data show that those who finish closer or even after the due date perform more poorly than those who make their last submission earlier?

Figure 4-16 and Figure 4-17 show grade information for the Change Maker assignment. Looking at Figure 4-16, we can see that almost all of the students who started before the due date, received a perfect score. We can also see that the majority of those who did not start until

after the due date did not receive a perfect score. It is important to note here that there are late points associated with turning the assignment in late. It is hard to discern whether the cause for the lower scores is only attributed to the late points or the fact that the assignment was not completed fully. Looking at Figure 4-17, we can draw similar conclusions. Those who finished early received the highest grades. Those who finished later, mainly after the deadline, received lower grades, but as mentioned earlier it is dififcult to attribute the lower scores to poorer performance or to simply the late points. Note here that the highest grade is a 95 due to 5 points being allocated to a pre-homework quiz. These observations make sense for an assignment of lesser difficulty, but what about for an assignment of higher difficulty like the Tweets assignment? Looking at Figure 4-18 and Figure 4-19, we can see the same graphs, but for the Tweets assignment. In general, those that started earlier received better grades than those who started later. As for grade versus the time of the last submission, no trends can automatically be identified since the variance is so much higher than the Change Maker assignment.

These are just a couple of examples of the optional data visualizations that PRAISE supports. PRAISE also offers visualizations that correlate the Mimir data with attendance, exam grades, and Piazza participation. In addition to viewing a single assignment, we can also support viewing graphs for multiple assignments side-by-side, which may be beneficial for identifying

overall trends for assignments as the difficulty changes. We also support viewing assignments across semesters so that comparisons can be made from one semester to the next.

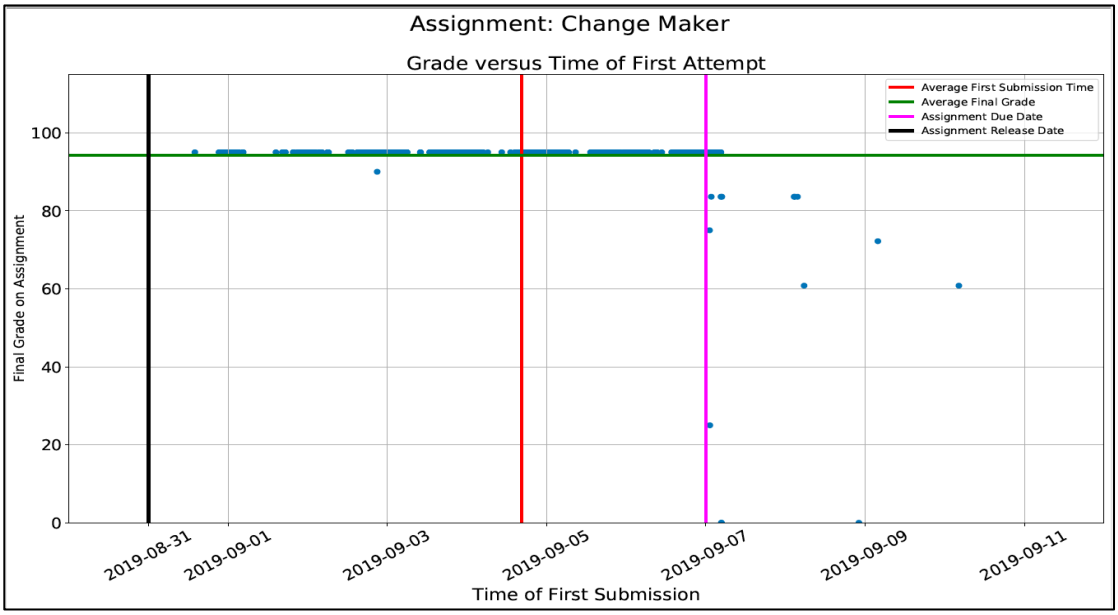


Figure 4-16 Graph displays grade versus time of the first submission for each student for the Change Maker assignment.

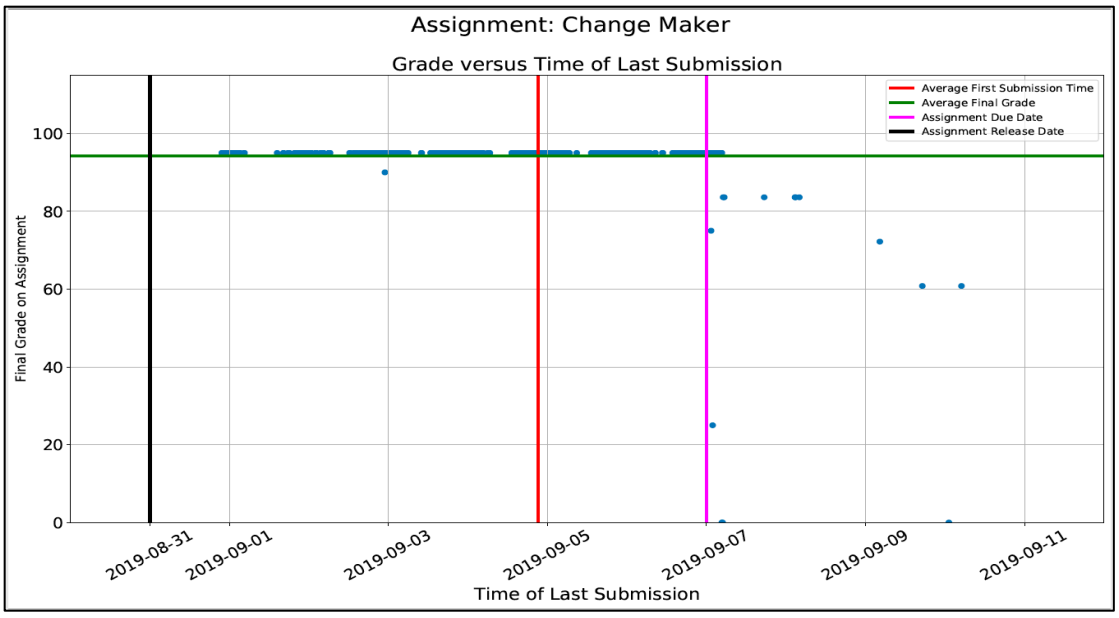


Figure 4-17 Graph displays the assignment grade versus the time of the last submission for each student for the Change Maker assignment.

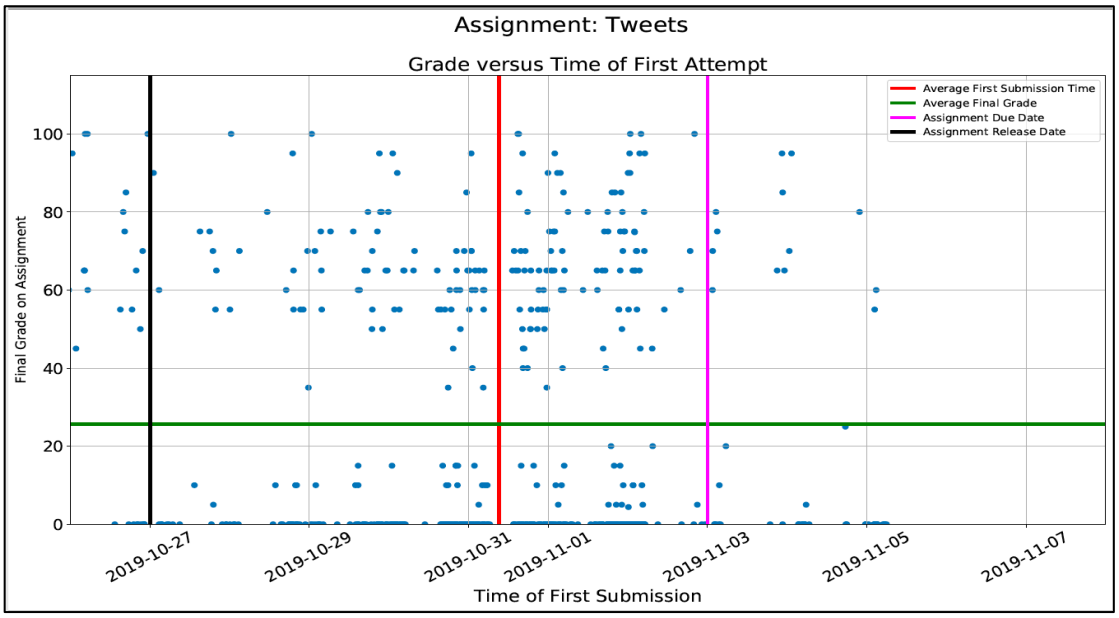


Figure 4-18 Graph displays the assignment grade versus the time of the first submission for each student for the Tweets assignment.

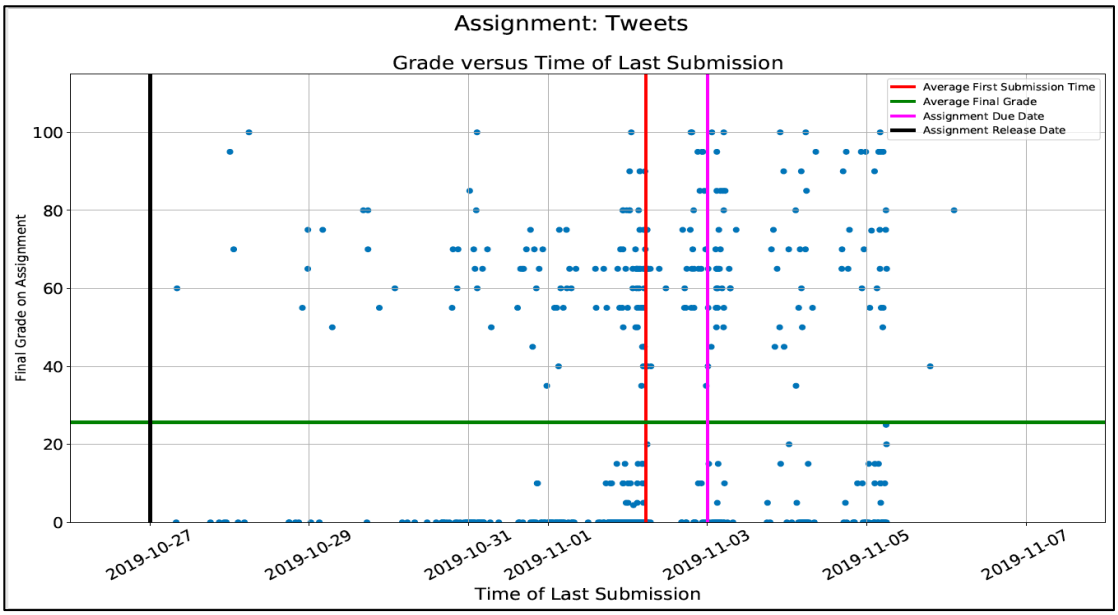


Figure 4-19 Graph displays grade versus time of the last submission for each student for the Tweets assignment

4.3 Plagiarism Detection Enhancement

Before discussing how PRAISE can enhance plagiarism detection, we report in our experience using MOSS to flag plagiarism. We obtained MOSS reports for all ten homeworks in the Fall 2019 offering, and we discuss here the results for the Change Maker Tweets assignments. As discussed in Chapter 1, code similarity tools may be ineffective with introductory assignments. We found that a 99 percent similarity on a simple assignment like Change Maker is a lot different than a 99 percent similarity on a more complicated and complex assignment, like Tweets. The instances where the similarity reported by MOSS was above 90 percent on the Tweets assignment were easy to identify as plagiarism just by inspecting the code. On the other hand, even a 98 percent similarity, the highest similarity for the Change Maker

assignment, could not be conclusively identified as plagiarism by inspecting the code. Figures 4.20 and 4.21 show the top part of the MOSS report, with student e-mails omitted.

After labeling the instances from the MOSS report as plagiarism or not based on our code inspection, we ran the Moss_Analysis script to enhance the MOSS report with the submission pattern data we extracted from Mimir.

When looking at the Tweets assignments and the top submissions labeled as plagiaristic, we found some interesting patterns. The MOSS report is useful because it makes it easy to identify students who worked together as a group. Although we can use the MOSS report as a guide to identifying students who worked together, the MOSS information does not provide any evidence of who may have originated the base solution for the group source code copy. With the information provided by PRAISE, there may be evidence that one of the students worked by incrementally submitting code versions until the solution worked, and the other students simply started from a copy of that working solution. This information does not change things in terms of plagiarism; given the clear syllabus specification that no collaboration is allowed on the assignments, all students, regarding who may have written the code originally, may be have violated the honor code for the course. Nonetheless, the instructor may gain insights on overall student learning by considering how each of the students involved in high-similarity code submission engaged in the homework assignment.

We found that the information gathered by PRAISE can be useful in providing information about plagiarism scenarios where code has been copied without the knowledge or the consent of its author. The submission patterns can reveal that one student made incremental progress towards a solution, and had no history of code similarity with other students in the other assignments.

Moss Results

Fri Nov 29 09:30:10 PST 2019

Options -l cc -m 100

[How to Read the Results | Tips | FAQ | Contact | Submission Scripts | Credits]

File 1	File 2	Lines Matched
XYZ@tamu.edu-coinmaker.cpp (91%)	XYZ@tamu.edu-coinmaker.cpp (94%)	45
XYZ@tamu.edu-coinmaker.cpp (76%)	XYZ@tamu.edu-coinmaker.cpp (77%)	30
XYZ@tamu.edu-coinmaker.cpp (94%)	XYZ@tamu.edu-coinmaker.cpp (96%)	50
XYZ@tamu.edu-coinmaker.cpp (76%)	XYZ@tamu.edu-coinmaker.cpp (76%)	32
XYZ@tamu.edu-coinmaker.cpp (77%)	XYZ@tamu.edu-coinmaker.cpp (76%)	30
XYZ@tamu.edu-coinmaker.cpp (98%)	XYZ@tamu.edu-coinmaker.cpp (98%)	33
XYZ@tamu.edu-coinmaker.cpp (63%)	XYZ@tamu.edu-coinmaker.cpp (64%)	26
XYZ@tamu.edu-coinmaker.cpp (52%)	XYZ@tamu.edu-coinmaker.cpp (67%)	24
XYZ@tamu.edu-coinmaker.cpp (68%)	XYZ@tamu.edu-coinmaker.cpp (52%)	16
XYZ@tamu.edu-coinmaker.cpp (37%)	XYZ@tamu.edu-coinmaker.cpp (44%)	33
XYZ@tamu.edu-coinmaker.cpp (57%)	XYZ@tamu.edu-coinmaker.cpp (65%)	23
XYZ@tamu.edu-coinmaker.cpp (50%)	XYZ@tamu.edu-coinmaker.cpp (66%)	23
XYZ@tamu.edu-coinmaker.cpp (89%)	XYZ@tamu.edu-coinmaker.cpp (89%)	24
XYZ@tamu.edu-coinmaker.cpp (49%)	XYZ@tamu.edu-coinmaker.cpp (65%)	25
XYZ@tamu.edu-coinmaker.cpp (49%)	XYZ@tamu.edu-coinmaker.cpp (64%)	28
XYZ@tamu.edu-coinmaker.cpp (87%)	XYZ@tamu.edu-coinmaker.cpp (83%)	23
XYZ@tamu.edu-coinmaker.cpp (63%)	XYZ@tamu.edu-coinmaker.cpp (58%)	21
XYZ@tamu.edu-coinmaker.cpp (65%)	XYZ@tamu.edu-coinmaker.cpp (59%)	18
XYZ@tamu.edu-coinmaker.cpp (41%)	XYZ@tamu.edu-coinmaker.cpp (63%)	18
XYZ@tamu.edu-coinmaker.cpp (69%)	XYZ@tamu.edu-coinmaker.cpp (65%)	28
XYZ@tamu.edu-coinmaker.cpp (60%)	XYZ@tamu.edu-coinmaker.cpp (48%)	24
XYZ@tamu.edu-coinmaker.cpp (60%)	XYZ@tamu.edu-coinmaker.cpp (41%)	21
XYZ@tamu.edu-coinmaker.cpp (63%)	XYZ@tamu.edu-coinmaker.cpp (62%)	21
XYZ@tamu.edu-coinmaker.cpp (48%)	XYZ@tamu.edu-coinmaker.cpp (55%)	25
XYZ@tamu.edu-coinmaker.cpp (61%)	XYZ@tamu.edu-coinmaker.cpp (62%)	20
XYZ@tamu.edu-coinmaker.cpp (47%)	XYZ@tamu.edu-coinmaker.cpp (61%)	27
XYZ@tamu.edu-coinmaker.cpp (74%)	XYZ@tamu.edu-coinmaker.cpp (74%)	34
XYZ@tamu.edu-coinmaker.cpp (72%)	XYZ@tamu.edu-coinmaker.cpp (73%)	28
XYZ@tamu.edu-coinmaker.cpp (71%)	XYZ@tamu.edu-coinmaker.cpp (71%)	43
XYZ@tamu.edu-coinmaker.cpp (67%)	XYZ@tamu.edu-coinmaker.cpp (73%)	28
XYZ@tamu.edu-coinmaker.cpp (60%)	XYZ@tamu.edu-coinmaker.cpp (53%)	16
XYZ@tamu.edu-coinmaker.cpp (46%)	XYZ@tamu.edu-coinmaker.cpp (47%)	19
XYZ@tamu.edu-coinmaker.cpp (56%)	XYZ@tamu.edu-coinmaker.cpp (47%)	17
XYZ@tamu.edu-coinmaker.cpp (78%)	XYZ@tamu.edu-coinmaker.cpp (80%)	32
XYZ@tamu.edu-coinmaker.cpp (58%)	XYZ@tamu.edu-coinmaker.cpp (33%)	23
XYZ@tamu.edu-coinmaker.cpp (57%)	XYZ@tamu.edu-coinmaker.cpp (58%)	27
XYZ@tamu.edu-coinmaker.cpp (81%)	XYZ@tamu.edu-coinmaker.cpp (81%)	20
XYZ@tamu.edu-coinmaker.cpp (59%)	XYZ@tamu.edu-coinmaker.cpp (52%)	18
XYZ@tamu.edu-coinmaker.cpp (63%)	XYZ@tamu.edu-coinmaker.cpp (68%)	34
XYZ@tamu.edu-coinmaker.cpp (45%)	XYZ@tamu.edu-coinmaker.cpp (55%)	18
XYZ@tamu.edu-coinmaker.cpp (53%)	XYZ@tamu.edu-coinmaker.cpp (58%)	22

Figure 4-20 Anonymized MOSS report for Change Maker assignment

Moss Results

Sat Nov 30 08:01:25 PST 2019

Options -l cc -m 100

[How to Read the Results | Tips | FAQ | Contact | Submission Scripts | Credits]

File 1	File 2	Lines Matched
XYZ@tamu.edu-functions.cpp (87%)	XYZ@tamu.edu-functions.cpp (90%)	257
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	197
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	154
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	197
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	155
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	197
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	155
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	154
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	197
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	154
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	155
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	154
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	197
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	154
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	155
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	154
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	156
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	156
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	156
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	157
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	157
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (98%)	157
XYZ@tamu.edu-functions.cpp (97%)	XYZ@tamu.edu-functions.cpp (98%)	157
XYZ@tamu.edu-functions.cpp (97%)	XYZ@tamu.edu-functions.cpp (98%)	198
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (97%)	157
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (97%)	198
XYZ@tamu.edu-functions.cpp (97%)	XYZ@tamu.edu-functions.cpp (98%)	157
XYZ@tamu.edu-functions.cpp (97%)	XYZ@tamu.edu-functions.cpp (98%)	157
XYZ@tamu.edu-functions.cpp (97%)	XYZ@tamu.edu-functions.cpp (98%)	157
XYZ@tamu.edu-functions.cpp (97%)	XYZ@tamu.edu-functions.cpp (98%)	157
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (97%)	157
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (97%)	198
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (97%)	157
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (97%)	157
XYZ@tamu.edu-functions.cpp (98%)	XYZ@tamu.edu-functions.cpp (97%)	158

Figure 4-21 Anonymized MOSS report for the Tweets assignment

When inspecting the MOSS report for the Tweets assignment (Figure 4-21), we identified a large number of students who had submissions with 98 or 99 percent similarity. As discussed above, based on just the MOSS report, we could only identify what group of students all had the same solution, but could not identify where the solution most likely originated from. With the features provided by PRAISE, we could easily identify which student from the group made their last submission first and analyze his/her submission pattern. One of the students made their last

submission at least 2 days before the first submission time of the other students in this group. This student also made a total of 9 submissions; whereas, the others in the group made 5 or less submissions, except for one student. Also, several of the other students made relatively large changes to the code on their last submission.

Looking at another group of students, about 15 students, who all had the same solution, no student immediately stood out as having finished much earlier than everyone. After taking a close look, the information in PRAISE allowed us to identify the student who made their last submission before everyone else. That student made a total of 49 submissions. Each of the other students in this group made 12 or fewer submissions. A few of these students made relatively large changes to the code on their last submission. We could also easily see that the majority of these students started the assignment late, submitting it right before or even after the deadline.

These experiences indicate that the PRAISE framework, by adding features extracted from the the Mimir submission history, can make identifying the source of the plagiarized code easier. For situations where the source of the common solution is not a student (e..g, a tutoring service), PRAISE's ability to carry out data analysis across different offerings (different instructors reusing an assignment, possibly in different semesters) can bring a great advantage to the plagiarism detection effort in large departments, where thousands of students may take the same course every semester. The data confirmed our intuition that, in general, those who plagiarize using another student's code tend to have fewer submissions than the original student. Also, those that plagiarize finish and often start after the original student has finished submitting their code. This may be because the original student has first worked through multiple submissions to pass all test cases required for the homework, and then they pass on their working solution to others. We also have observed large changes in the last submission by those who

plagiarized. All of these observations were made by looking at the data presented by PRAISE by extracting features from the Mimir submissions.

	Student 1	Student 2
Total Submissions (Percentile)	3 (68.7976)	1 (16.5145)
% Lines Changed on Last Submission (Percentile)	0 (31.422)	100 (31.4220)
Days Between First and Last Submission (Percentile)	0.0112 (64.3951)	0 (16.6922)
First Submission Time	9/2/19 12:10 AM	9/3/2019 5:20 PM
Last Submission Time	9/2/19 12:26 AM	9/3/2019 5:20 PM

Table 4-1 Extracted Mimir Metrics for the 98 Percent Similarity Instance for the Change Maker Assignment.

As mentioned previously, for simple assignments, not even a 99 percent similarity can be used conclusively as evidence of plagiarism due to the inherent limited variaions in expressing simple tasks as code in C++. Also, in simple assignments the style and approach used by the instructor in the classroom may lead to many students developing code with the same structure and the same approach for other characteristics such as the choice of variable names. Considering this, PRAISE can aid in identifying cases of plagiarism by exposing information on how students worked to derive the submitted solutions . For example, from the MOSS report for the Change Maker assignment shown in Figure 4-20, we have an instance where 2 students share

code that is 98 percent similar, but since there are so few ways for completing the assignment, simply looking at their code was not sufficient for us, as instructors, to confirm plagiarism. With the additional Mimir features, we gain additional insight. Table 4-1 shows those features for the two students from line 6 of Figure 4-20. We can see that Student 1 submitted a total of 3 times, with no lines changed for their last submission, and a short time between their first and last submission. Student 2 submitted only once with their last submission being a full day after Student 1. Taking all of this into consideration, we can say that this is a probable instance of plagiarism that should be further investigated.

	Student 1	Student 2
Total Submissions (Percentile)	4 (81.3546)	2 (46.8037)
% Lines Changed on Last Submission (Percentile)	0 (31.422)	12.5 (95.1835)
Days Between First and Last Submission (Percentile)	0.1766 (86.6003)	0.0036 (34.8392)
First Submission Time	9/4/2019 12:26 AM	9/3/2019 6:09 PM
Last Submission Time	9/4/2019 4:40AM	9/3/2019 6:14PM

Table 4-2 Extracted Mimir Metrics for the 91 Percent Similarity Instance for the Change Maker Assignment.

Table 4-2 shows the Mimir features for another example where the MOSS report reported two students having 91 percent similarity (line 1 of Figure 4-X). As with the first example, looking at the code for each student and the MOSS report is not conclusive. The PRAISE data shows that the Student 2 submitted twice and made their last submission before Student 1 started, who made 4 attempts. The PRAISE information on their own may be indicative of plagiarism, but are not conclusive of plagiarism, just like the MOSS report on code similarity and code inspections are not definitely conclusive of plagiarism. Even when we consider the PRAISE information, including the Mimir submission pattern features, in combination with the code of the students, we can not conclude if they are plagiarized or not, just similar. We expect to see similar approaches to simple assignments since there is a limited number of ways to complete them. In situations like these, the information provided by PRAISE can clear up much of the ambiguity associated with identifying plagiarism of simpler assignments. PRAISE can help the instructors to distinguish the cases that are more likely to be plagiarism, helping them to prioritize their investigation effort to address first the scenarios of more likelihood of plagiarism.

If we look at the instances on the MOSS reports that were labeled as not being plagiarized, we can identify some characteristics that are different than those discussed earlier. The students who did not plagiarize generally have more submissions than those who did plagiarize. Instances of only one submission are almost non-existant among those who did not plagiarize. Also, for the most part, the time between a student's first submission and last submission is greater than those who plagiarized. We also can see that, generally, those that plagiarize start later, closer to the deadline than those who do not plagiarize. The fact that we can identify different characteristics for cases of plagiarism and those that did not plagiarize is indicative of the fact that the Mimir features do enhance plagiarism detection.

5. CONCLUSIONS

5.1 Summary

We have designed PRAISE (Programming Activity Indicators of Student Effort) as a framework for pulling together student and course data from several different sources. PRAISE takes this data and visualizes it to give more insight into overall course trends and individual student programming fingerprints. The current version of PRAISE brings together data from submissions on homework assignments and labs, attendance, exam grades, and Piazza participation.

We have found that PRAISE's data visualizations aid in better understanding the coding behaviors of the class as a whole and on a more individual level. Generally, when we view data from each of the available sources on their own, trends are hard to identify and understand. PRAISE helps to eliminate this issue by creating a suite where all data can be analyzed and visualized side-by-side. We also make it possible to view data across assignments and then across semesters. This gives the user an even deeper understanding of the course as a whole. The data repository can also enable additional data mining that may lead to new insights as the instructor and PRAISE administrators pursue new questions.

In addition to the data visualization, PRAISE provides an enhancement of the traditional plagiarism detection. Plagiarism is complicated. Tools like MOSS offer insight into the similarity of code, but it is far away from providing conclusive data. Our framework can assist in this by enhancing the code similarity indices with additional features. We capture in PRAISE features related to the process of developing assignment solutions. By observing these features, an instructor gains insight into the coding behaviors of a student. Certain behaviors are more

indicative of plagiarism than others, so combining such information with code similarity rankings enhances the process of plagiarism detection.

Our framework is just in the beginning stages. We expect PRAISE to evolve and change as instructors and computer science (CS) education researchers gain experience with obtaining data to reveal the efforts students dedicate to their activities. In its current state, PRAISE already provides informative visualizations and assistance with plagiarism detection. PRAISE has been designed to be expanded and improved so that it becomes an impactful toolset that instructors can utilize to better understand courses and students.

5.2 Conclusions

PRAISE shares some similarities with those mentioned in the related work. For example, the AC [10,11] plagiarism detection system was found to be advantageous because it produces helpful visualizations even when there are a large number of submissions. AC focuses on a distance measure between submissions to identify and visualize plagiarism. PRAISE offers visualizations that provide insight into the overall submission trends. This helps to identify outlying behavior that may be indicative of plagiarism.

PRAISE also captures student coding behaviors by extracting information about their submission habits. We extract information about each individual submission each student makes to Mimir. TMOSS [16] does something similar by periodically uploading student submissions to MOSS [9], so that intermediate MOSS reports can be captured in order to better identify plagiarism. PRAISE enhances the traditional approach to plagiarism detection using MOSS and gives further insight into overall submission trends. PRAISE is different than MOSS and TMOSS because it extracts additional features about the student fingerprint not given by these

systems. We gather timestamp information, percentage of lines changed on each submission, total number of submissions, and all of the relative percentiles.

Compared to Smith's work of trying to identify and understand a student's coding fingerprint by looking at their GitHub commits [6], we go a step further. The previous work done to capture a student's coding fingerprint offered some unique visualizations and insight into overall trends, but was not integrated with any plagiarism detection system. We expanded on the ideas in Smith's research in order to enhance plagiarism detection. Another benefit from PRAISE is the fact that it works with the information available from an autograding system such as Mimir. In many institutions, students have to use an autograding system to submit their work, receive feedback, and get their assignments graded. This is advantageous because we can still capture student's coding fingerprint without imposing additional requirements such as using a version control system such as Github, which can be quite complicated for beginners.

In conclusion, we envision PRAISE as the foundation for something bigger. We showed that PRAISE can harness Mimir submission information to improve plagiarism reports. PRAISE can also help instructors to gain a deeper understanding of class trends and individual student coding behavior. In addition to general course information, PRAISE can enhance traditional plagiarism detection methods. This framework has the potential to evolve into a complete interactive toolset for instructors and researchers, serving as a platform for data analysis of courses and plagiarism detection.

5.3 Future Work

This research is just the initial step to building something better that provides even more detailed information to instructors regarding the student coding fingerprint. We believe that as

we build on this framework, we will be able to identify and incorporate additional useful information. For example,

at the moment we only correlate the Mimir metrics with the assignment grade. We would like to extrend on this and correlate the Mimir metrics with the overall course grade and exam grades. This is just an example of the additional information we plan to provide.

This research led to an initial framework prototype to advance plagiarism detection, a common problem in computer science courses. This framework can be expanded to further support instructors and researchers in data analysis of the vast amount of data we have access to in courses like CSCE 121. As mentioned earlier, the current data store is a set of csv files, which is not scalable. For the target data exploration of this research, the csv file data store was sufficient, but to expand PRAISE towards its potential, a more scalable and resilient approach for data storage and manipulation will need to be implemented. A first, immediate step in the future work would be to migrate to a database structure in order to allow for scalability across more courses and course offerings.

The current PRAISE prototype supports plagiarism detection by enhancing the MOSS report to include data about student submission patterns when submitting solutions to an autograding environment such as Mimir. We plan to improve such support by introducing a ranking system that predicts the likelihood that submissions flagged by code similarity measures as plagiarism are, indeed, plagiarism. The ranking system will incorporate machine learning methods to model student behavior. This would require obtaining ground truth data, i.e., some set of training data where cases of plagiarism have been properly identified by experts (instructors). At this point, we believe that such time-consuming task would be necessary for any new assignment being added to PRAISE, but we may find that with appropriate characterization

of assignments, existing models can be applied to new assignments without acquisition of more training data. Regardless, such effort by course staff would greatly improve the trustworthiness of our plagiarism ranking system. In addition to creating the training data, evaluation of the models and testing them would need to be done to ensure that the ranking system is trustworthy and behaves as intended.

PRAISE would benefit from a well-designed user interface (UI.) The UI could guide a user through what files are needed and any optional features they may be interested in viewing. A UI that allows the graphs and visualizations to be interacted with would be ideal. Allowing the users to select ranges of data, easily switch features, easily compare to similar assignments, etc. are all things that need to be addressed in the future.

Taking PRAISE even further, we would like to analyze submissions to identify where students may be struggling. The goal is to identify what issues are common among a group of students, so that those issues can be addressed through autograding feedback, labs, or class discussion. We envision PRAISE as advancing the state-of-art in providing formative feedback to students by removing some of the limitation present in efforts such as [30]. A feature like this could be harnessed in order to improve overall performance and understanding for the students in the course.

In the future, we plan to deploy PRAISE throughout the entire semester rather than just at the end of the semester like we have done for the Fall 2019 CSCE 121 offering. Instructors will be able to identify plagiarism early in the semester. This will allow the teaching staff to identify students who resort to plagiarism because they are struggling. If those students can be identified and helped early on, they may be able to succeed in the course. It is also important to identify plagiarism early in order to penalize those who do it so that the word spreads and students are

less likely to do it in the future. We want to help the students to succeed on their own, so they have a better chance of success in higher-level courses.

REFERENCES

[1] J. Bidgood and J. B. Merrill. “As Computer Coding Classes, So Does Cheating,” *New York Times,* p. A1, 29-May-2017

[2] Martin Potthast, Benno Stein, Alberto Barrón-Cedeño, and Paolo Rosso. 2010. An evaluation framework for plagiarism detection. In Proceedings of the 23rd International Conference on Computational Linguistics: Posters (COLING '10). Association for Computational Linguistics, Stroudsburg, PA, USA, 997-1005.

[3] Eissen S.M.., Stein B. (2006) Intrinsic Plagiarism Detection. In: Lalmas M., MacFarlane A., Rüger S., Tombros A., Tsikrika T., Yavlinsky A. (eds) Advances in Information Retrieval. ECIR 2006. Lecture Notes in Computer Science, vol 3936. Springer, Berlin, Heidelberg

[4] Stack Overflow, https://stackoverflow.com

[5] RosettaCode, http://www.rosettacode.org/wiki/Rosetta_Code

[6] C. M. Smith, "A Toolset for Mining GitHub Repositories in Educational Software Projects," M. S. thesis, Dept. of Computer Sc. and Eng., Texas A&M Univ., College Station, TX, 2018.

[7] Karl J Ottenstein. 1976. An algorithmic approach to the detection and prevention of plagiarism. ACM Sigcse Bulletin 8, 4 (1976), 30–41.

[8] H. T. Jankowitz, Detecting Plagiarism in Student Pascale Programs, *The Computer Journal*, Volume 31, Issue 1, 1988, Pages 1–8, https://doi.org/10.1093/comjnl/31.1.1

[9] Alex Aiken. 2004. MOSS: A System for Detecting Software Plagiarism. https://theory.stanford.edu/~aiken/moss/ (2004).

[10] M. Freire. Visualizing Program Similarity in the AC Plagiarism Detection System. In Proceedings of the Working Conference on Advanced Visual Interfaces, AVI ’08, pages 404–407, New York, NY, USA, 2008. ACM.

[11] M. Freire, M. Cebrian, and E. Rosal. AC: An Integrated Source Code Plagiarism Detection Environment. Technical Report cs.IT/0703136, Universidad Aut´onoma de Madrid, Mar 2007. Comments: 57 pages, 11 figures.

[12] B. Zeidman. Tools and algorithms for finding plagiarism in source code. Dr Dobbs, July 01 2004.

[13] T. Copeland. Detecting duplicate code with pmd’s cpd, Dec 03 2001.

[14] B. Haskins. Utilising n-grams and Edit Distance as a Means of Identifying Copied Programming Assignments. In Proceedings of the 44th Annual Conference of the Southern African Computer Lecturers’ Association (SACLA), Port Elizabeth, 25 -26 June 2014. SACLA Organising Committee.

[15] Phatludi Modiba, Vreda Pieterse, and Bertram Haskins. 2016. Evaluating plagiarism detection software for introductory programming assignments. In Proceedings of the Computer Science Education Research Conference 2016 (CSERC '16), Vreda Pieterse and Marko van Eekelen (Eds.). ACM, New York, NY, USA, 37-46. DOI: http://dx.doi.org/10.1145/2998551.2998558

[16] 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 (SIGCSE '18). ACM, New York, NY, USA, 110-115. DOI: https://doi.org/10.1145/3159450.3159490

[17] Narjes Tahaei and David C. Noelle. 2018. Automated Plagiarism Detection for Computer Programming Exercises Based on Patterns of Resubmission. In Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18). ACM, New York, NY, USA, 178-186. DOI: https://doi.org/10.1145/3230977.3231006

[18] Jonathan Pierce and Craig 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. DOI: https://doi.org/10.1145/3017680.3017797

[19] N. Fonseca, L. Macadeo, & A. Mendes . 2018. Using early plagiarism detection in programming classes to address the student's difficulties. 2018 International Symposium on Computers in Education (SIIE). Jerez, Spain. DOI: 10.1109/SIIE.2018.8586700.

[20] Generation CS: Report on CS Enrollment. *CRA*, 2019. https://cra.org/data/generation-cs/.

[21] Growth of Computer Science Undergraduate Enrollments. *Sites.nationalacademies.org*, 2019. https://sites.nationalacademies.org/CSTB/CompletedProjects/CSTB_171607.

[22] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 437-442. DOI: https://doi.org/10.1145/2839509.2844616

[23] J. McBroom, B. Jeffries, I. Koprinska, & K. Yacef. 2016, June. Mining behaviors of students in autograding submission system logs. In EDM (pp. 159--166)

[24] Vincent Gramoli, Michael Charleston, Bryn Jeffries, Irena Koprinska, Martin McGrane, Alex Radu, Anastasios Viglas, and Kalina Yacef. 2016. Mining autograding data in computer science education. In Proceedings of the Australasian Computer Science Week

Multiconference (ACSW '16). ACM, New York, NY, USA, , Article 1 , 10 pages. DOI=http://dx.doi.org/10.1145/2843043.2843070

[25] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. 2018. Providing Meaningful Feedback for Autograding of Programming Assignments. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18). ACM, New York, NY, USA, 278-283. DOI: https://doi.org/10.1145/3159450.3159502

[26] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In Proceedings of the 13th annual conference on Innovation and technology in computer science education (ITiCSE '08). ACM, New York, NY, USA, 328-328. DOI: https://doi.org/10.1145/1384271.1384371

[27] Mimir Classroom, https://www.mimirhq.com/classroom/competitive-comparison

[28] Vocareum Classroom, https://www.vocareum.com/#classroom

[29] Jupyter Notebook, https://jupyter.org/

[30] *Giving Hints is Complicated: Understanding the challenges of an automated hint system based on frequent wrong answers*. In Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education 2018. ACM ITiCSE '18.

APPENDIX A

HOMEWORK ASSIGNMENT GUIDELINES

A.1 Specification for the Change Maker Assignment

Individual Homework: Change Maker

Overview

You have been tasked to write a program that makes change with the minimum amount of coins for a given amount -- a 'Coin Maker', if you will. For example, $2.16 would consist of 8 quarters, 1 dime, 1 nickel, and 1 penny: 11 coins in total. Your code will be submitted to Mimir for autograding. Therefore, your code must match exactly the required input and output specified in the requirements.

Objectives

1. Exposure to reading and writing data via standard input and standard output respectively.
2. Familiarity with: the declaration of variables, reading and writing to the objects bound to them, and interacting with those named objects throughout program code
3. Experience working with the arithmetic types and the set of operations that can be performed on them.

Grading breakdown

Points	

5	Pre-Homework Quiz
95	Runs correctly (using instructor's test cases)
100	**TOTAL**

Requirements

When developing your solution to this problem, ensure that your program implements the following requirements:

- Name the source file containing the main function coinmaker.cpp.

- Source code is written such that it is readable by a novice programmer. Use descriptive variable identifiers and comments where appropriate (e.g. for non-trivial program code).

- Assume an unlimited number of 1¢, 5¢, 10¢, and 25¢ coins are available for change.

- Input is taken via standard input (i.e. cin): amount to make change from. It's never recommended to use floating-point numbers when dealing with monetary values; therefore, read in the amount to make change from as two integers value (one for dollars and another for cents)
    - You will first prompt for the whole-dollar amount and read that value.

- You will then prompt the user for the whole-number of cents and read that value

You can assume that the two integers provided as input are non-negative.

- If change is being made for less than a dollar (e.g. $0.24), assume that the user will always enter a zero for the whole-dollar amount. If change is being made for a whole-dollar amount with no cents (e.g. $12.00), assume that the user will always enter zero for the whole-number of cents.

- Makes change with the minimum amount of coins for a given amount. Think about how you can accomplish this using modulo (%) and division (/).

- Output via standard output (i.e. cout): number of coins of each denomination and the total number of coins.

- Required format of i/o (here we use $12.34 as example input; user input in bold **red**; everything else is output):

Enter dollars: **12**↲

Enter cents: **34**↲

Pennies: 4↲

Nickels: 1↲

Dimes: 0↲

Quarters: 49↲

↵

Total coins used: 54↵

A.2 Specification for the Tweets Assignment

Individual Homework: Twitter Analytics

Objectives

- Work with C++ strings. In the supporting information, we list a few string methods that you may find useful.
- Work with reading strings from files.
- Work with function arguments passed by reference.
- Further practice with dynamic allocation and resizing of arrays.
- Further practice with statically-allocated arrays.
- Further practice with accumulating data statistics.
- Further practice with the process of fixing code errors. If you need motivation, you may want to look at a few quotes about bugs and debugging at the end of this document.

Grading breakdown

Points	
100	Runs correctly
10	Bonus: Early-bird submission
110	**MAX TOTAL POINTS**

Overview

We all know companies that dedicate entire data centers with hundreds of thousands of machines to the analysis of social media data. In many cases, these machines are high-cost ones due to

their large-memory configuration (current prices are around $ 20,000 for a machine with 2 TB of RAM.)

You were hired as an intern by a startup that is working hard to raise money from venture capitalists, but until they have enough cash to buy equipment, they need you to write data analytics code that can run on the computers that their interns bring with them for the internship. One of the company founders was your TA in CSCE 121 and is aware that you have experience with dynamically resizing arrays and paying attention to memory leaks.

The company wants to offer a new service to their customers: low-cost statistics on the usage of hashtags on Twitter[2] data streams. They want you to create a program that customers can use to upload twitter data and get statistics about the most popular hashtags or statistics about a hashtag of particular interest to the customer. Tweet data files can be quite large: users produce 500 million tweets per day and around 200 billion tweets per year[3]. Due to the limited availability of memory in the company machines, your program must analyze tweets, update statistics about hashtag popularity accordingly, and then discard the tweet. The screenshots below show how users will interact with your program (user input appears in **red**.)

First, the user runs the program and sees the menu of options, selecting option 1 and providing the name of the file containing tweet data:

```
Welcome to Aggieland Twitter Feeds Stats
```

[2] Twitter is a microblogging and social network service founded in 2006 [https://en.wikipedia.org/wiki/Twitter]
[3] Source: https://www.internetlivestats.com/twitter-statistics/, visited in October 24, 2019.

```
The options are:

1. load tweet data file and update stats

2. show overall stats (number of tweets, retweets, and hashtags)

3. show most popular hashtags

9. quit

--------> Enter your option: 1

Enter filename: TAMU-small.csv


Welcome to Aggieland Twitter Feeds Stats

The options are:

1. load tweet data file and update stats

2. show overall stats (number of tweets, retweets, and hashtags)

3. show most popular hashtags

9. quit

--------> Enter your option:
```

Now the user asks for option 2 (show stats) and option 3 (show at most 10 most popular tweets so far):

```
--------> Enter your option: 2

Tweets: 3, Retweets: 2, Hashtags: 5


Welcome to Aggieland Twitter Feeds Stats

The options are:
```

```
1. load tweet data file and update stats

2. show overall stats (number of tweets, retweets, and hashtags

3. show most popular hashtags

9. quit

--------> Enter your option: 3

Tag #tamu - 2 occurrence(s)

Tag #aggieland - 1 occurrence(s)

Tag #gigem - 1 occurrence(s)

Tag #aggienetwork - 1 occurrence(s)

Tag #sec - 1 occurrence(s)
```

As you can see, the TAMU-small.csv file is a very small one, useful for debugging code.

The user can keep uploading new files. Now a larger one:

```
--------> Enter your option: 1

Enter filename: starwars.csv


Welcome to Aggieland Twitter Feeds Stats

The options are:

1. load tweet data file and update stats

2. show overall stats (number of tweets, retweets, and hashtags)
```

```
3. show most popular hashtags

9. quit

--------> Enter your option: 3

Tag #starwars - 13477 occurrence(s)

Tag #theriseofskywalker - 5162 occurrence(s)

Tag #funko - 3838 occurrence(s)

Tag #pop - 3828 occurrence(s)

Tag #e - 2104 occurrence(s)

Tag #giveaway - 1753 occurrence(s)

Tag #the - 1720 occurrence(s)

Tag #ep9 - 1714 occurrence(s)

Tag #starwarstheriseofskywalker - 835 occurrence(s)

Tag #returnofthejedi - 739 occurrence(s)
```

The user again asks to load another file and look at the stats:

```
--------> Enter your option: 1

Enter filename: frozen2.csv



Welcome to Aggieland Twitter Feeds Stats

The options are:

1. load tweet data file and update stats

2. show overall stats (number of tweets, retweets, and hashtags)

3. show most popular hashtags
```

9. quit

--------> Enter your option: **3**

Tag #frozen2 - 15963 occurrence(s)

Tag #starwars - 13488 occurrence(s)

Tag #theriseofskywalker - 5175 occurrence(s)

Tag #funko - 3888 occurrence(s)

Tag #pop - 3865 occurrence(s)

Tag #e - 2366 occurrence(s)

Tag #giveaway - 1764 occurrence(s)

Tag #the - 1720 occurrence(s)

Tag #ep9 - 1714 occurrence(s)

Tag #choprasisters - 1015 occurrence(s)

Welcome to Aggieland Twitter Feeds Stats

The options are:

1. load tweet data file and update stats

2. show overall stats (number of tweets, retweets, and hashtags)

3. show most popular hashtags

9. quit

--------> Enter your option: **2**

Tweets: 47423, Retweets: 30460, Hashtags: 3007

The input files have real data. They were obtained by connecting to a Twitter API (application programming interface) and invoking services that retrieve small collections of recent tweets. These input files (and many others) are available in the Google Drive (folder datasets).

The input files contain one tweet per line. The tweets may contain characters such as emoticons that do not display well as a text file. As an example, the file TAMU-small.csv contains only three tweets:

```
2019-10-22 14:57:03,"Are you a coffee lover? Looking for a different spot to study?
 Not only are there places to get coffee on campus, b… https://t.co/6p0OGezy8C"
2019-10-22 14:56:43,"RT @TAMU: Technology created by Aggies at @TAMUEngineering has
 the potential to save millions of lives! #tamu
2019-10-22 15:51:02,RT @JSH8_8: 2nd Official visit in College Station!👍■☑  #Aggie
land #GIGEM  #TAMU #AggieNetwork #SEC https://t.co/jjwJlGUKKT
```

You will need to process each tweet so that you extract the information you need: is it a retweet? What are its hashtags?

Requirements & Roadmap

To facilitate the integration of your code with other components of the system, the team leads at the company stated that your solution must use the declarations in the file functions.h. The file contains the declaration of the three functions that you must implement and use in your solution. These functions manipulate two structs that are also declared in functions.h (pictured in the next page.)

Notice the the struct OrderedHashtagList is dynamically allocated, growing as your system processes tweets and finds hashtags. As you add hashtags to the hashtag list, you may need to

allocate a larger area for the list by doubling the capacity list (like you did in the labwork in Week 8). You are asked to keep the list in decreasing order of hashtag popularity.

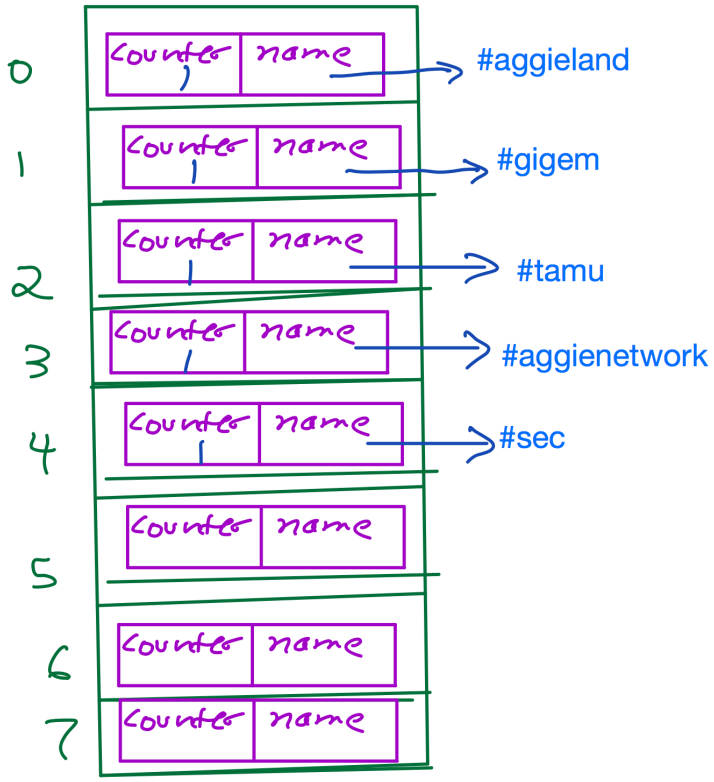
```
/* struct that keeps track of the statistics for a single
 * hashtag.
 */
struct Hashtag {
  std::string name;
  long counter = 0;     // total number of occurrences
};


/* struct that maintains a list of hashtags ordered in
 * decreasing order of popularity, i.e., in the
 * first position of the array we maintain the most popular hashtag.
 */
struct OrderedHashtagList {
  int capacity = 1;   /* how many positions have been allocated
                       * for the array 'list'. The 'list' array may be
                       * resized during
                       * execution, possibly growing or shrinking
                       */
  Hashtag* list = new Hashtag[capacity]; /* array of Hashtag elements
                                          * that must be kept sorted by
```

```
number
                                                * of occurrences of the hashtag.
                                                */
  int size = 0;  /* size of the array 'list', i.e., how many positions in
                        * the array are currently occupied
                        */
};
```

For example, if the first tweet that your system process has the following text:

```
2019-10-22 15:51:02,RT @JSH8_8: 2nd Official visit in College Station!👍■☑  #Aggie
land #GIGEM  #TAMU #AggieNetwork #SEC https://t.co/jjwJlGUKKT
```

After processing its hashtags, the variable holding your OrderedHashtagList will have the following data in its fields: capacity will be 8, size will be 5, and list points to an array of 8 Hashtag elements that resides in the heap:



If the next tweet processed has the hashtags #tamu and #awesomecsce121, then the counter for the #tamu hashtag will become 2, and as the now most popular hashtag, it will reside in position 0 of the array list. The hashtag is added to the end of the list with counter 1.

The functions that you are required to implement are:

void readTweet(string line, bool& isRetweet,

int& nb_htags, string*& array_of_htags);

Parameters:

- line: string containing the tweet information received by the function
- isRetweet: reference to bool; function will update it with true if retweet
- nb_htags: reference to int; function will update with number of hashtags in the tweet

- array_of_htags: reference to an array of strings; function will allocate the array and store the hashtags on it

Return value: none

Functionality: it processes the string in order to identify its hashtags and if it is a retweet. Real-life twitter data feeds present retweets in different ways. For the purpose of this homework, you only need to handle the following formatting for retweets:

- Appear preceded by ”

  2019-10-25 14:56:43,"RT @astudent: I will finish my csce 121 homework early”. Notice how it appears in code:

  string example(“\””)
- Appear preceded by comma, as in

  2019-10-25 18:14:40,RT @astudent: I am done with my homework
- Appear preceded by ‘

  2019-10-25 18:14:40,’RT @astudent: I dreamed I finished my homework

For hashtags, you can assume that they only contain letters and digits.

---

bool insertHashtag(string ht, OrderedHashtagList& hashlist);

Parameters:

- ht: string
- hashlist: reference to OrderedHashtagList struct

Return value: true if insertion succeeds, false if memory allocation fails.

Functionality: the function searches for the string ht in the hashlist's array. If the hashtag is already in the list, its counter is updated and the order of the elements in the array may need to be rearranged if the increment of the popularity counter changes the ordering of the hashtags.

If the hashtag is not in the list, it needs to be inserted. If there is no capacity in the array, a resize to double its capacity must be carried out.

In order to account for #TAMU and #tamu as the same hashtag, hashtags in the hashlist must be all in lowercase. The Useful functions section discusses the library support std::tolower and other methods that you may find useful when manipulating the string ht.

---

void showMostPopularHashtags(OrderedHashtagList hashlist, int k);

This function simply prints to the console the k most popular hashtags, i.e., the first k elements of the array in hashlist.

When invoking the function from the main program, you should use k = 10.

The main program and starter code

Your program needs to be able to open input files and read tweets lines from it for processing them. We provide starter code for tweets.cpp that prints the menu of options (function printMenu) and reads a user option from the console (function getOption)

The following screenshots show the format for a few error scenarios:

Welcome to Aggieland Twitter Feeds Stats

The options are:

1. load tweet data file and update stats

2. show overall stats (number of tweets, retweets, and hashtags)

3. show most popular hashtags

9. quit

--------> Enter your option: **5**

Invalid option

Welcome to Aggieland Twitter Feeds Stats

The options are:

1. load tweet data file and update stats

2. show overall stats (number of tweets, retweets, and hashtags)

3. show most popular hashtags

9. quit

--------> Enter your option: **d**

Invalid option

--------> Enter your option: **1**

Enter filename: **tammmmmmmuuuu.csv**

File can't be opened.

Welcome to Aggieland Twitter Feeds Stats

The options are:

```
1. load tweet data file and update stats
2. show overall stats (number of tweets, retweets, and hashtags
3. show most popular hashtags
9. quit
--------> Enter your option:
```

Notice the typo on error message “File can’t be open.”. It should be “File can’t be opened”, but we first released with this typo in the Mimir test case and changing it now would impact the students who finished the assignment already.

What to submit

You need to submit:

- functions.h
- functions.cpp (containing the three required functions and possibly other functions that you used in your solution)
- tweets.cpp (with the main program.)

The folder datasets in the Google Drive contains many datasets for you to use when running on your machine. All the datasets used in the Mimir tests are available.

Useful functions

You may use any of the functionality provided by C++ strings in your solution. Most students may find the following operations useful:

- compare or operator ==

  Example of reading from the console and comparing two strings:

```
int main() {
    string st1, st2;
    getline(cin, st1);
    getline(cin, st2);
    if (st1.compare(st2) == 0) {
            cout << "Strings are the same" << endl;
    } else {
            cout << "Strings are not the same" << endl;
    }
    if (st1 == st2) { // it is the same thing
            cout << "Strings are still the same" << endl;
    } else {
            cout << "Strings are still not the same" << endl;
    }
}
```

- find_first_of(char c): returns the position of the first occurrence of character c in the string. For example:

  string text(“CSCE121 is fun”);

  text.find_first_of(“1”) returns 4.

- find_first_of(char c, int pos): finds first position after pos. For example: text.find_first_of("1", 5) returns 6.
- std::isalpha(char c): returns true if c is an alphanumeric character
- std::isdigit(char c)
- std::tolower(char c): returns the lowercase form of char c

Change Log

- Version 1.1: Added comment on message error msg "file can't be open" - 10/27/19
- Version 1.0: initial release - 10/25/19

Quotes about testing and debugging (in case you are interested)

- "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence." (Edsger Dijkstra)
- "If debugging is the process of removing bugs, then programming must be the process of putting them in." (Edsger Dijkstra)
- "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." (Brian W. Kernighan)
- "A good programmer is someone who always looks both ways before crossing a one-way street." (Doug Linder)
- "Don't wait until you have a bug to step through your code" and "Never allow the same bug to bite you twice" (Steve Maguire, from his book "Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs")

- "With good program **design** debugging is a breeze, because bugs will be where they should be." (David May)
- “Without requirements or **design**, programming is the art of adding bugs to an empty text file.” (Louis Srygley)
- “When debugging, novices insert corrective code; experts remove defective code.” (Richard Pattis)
- “If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.” (Gerald Weinberg)
- “If McDonalds were managed like a software company, 1 out of every 100 Big Macs would give you food poisoning, and the response would be, ‘We’re sorry, here’s a coupon for two more.’ ” (Mark Minasi)
- “Where is the ‘any’ key?” (Homer Simpson, trying to figure out the “Press any key” notification)
- “There are two ways to write error-free programs; only the third one works.” (Alan J. Perlis)
- "Beware of bugs in the above code; I have only proved it correct, not tried it." (Donald E. Knuth)