From Clusters to Content: Using Code Clustering for Course Improvement

David A. Joyner
College of Computing
Georgia Institute of Technology
Atlanta, GA USA
david.joyner@gatech.edu

Ryan Arrison
College of Computing
Georgia Institute of Technology
Atlanta, GA USA
rarrison@gatech.edu

Mehnaz Ruksana
College of Computing
Georgia Institute of Technology
Atlanta, GA USA
mehnazruksana@gatech.edu

Evi Salguero
College of Computing
Georgia Institute of Technology
Atlanta, GA USA
esalguero3@gatech.edu

Zida Wang
College of Computing
Georgia Institute of Technology
Atlanta, GA USA
zwang795@gatech.edu

Ben Wellington
College of Computing
Georgia Institute of Technology
Atlanta, GA USA
bwellington3@gatech.edu

Kevin Yin
College of Computing
Georgia Institute of Technology
Atlanta, GA USA
kyi@gatech.edu

ABSTRACT

Large undergraduate CS courses receive thousands of code submissions per term. To help make sense of the large quantities of submissions, projects have emerged to dynamically cluster student submissions by approach for writing scalable feedback, tailoring hints, and conducting research. However, relatively little attention has been paid to the value of these tools for informing revision to core course materials and delivery methods. In this work, we applied one such technology—Sense, the eponymous product of its company—to an online CS1 class delivered simultaneously for credit to on-campus students and for free to MOOC students. Using Sense, we clustered student submissions to around 70 problems used throughout the course. In this work, we discuss the value of such clustering, the surprising trends we discovered through this process, and the changes made or planned to the course based on the results. We also discuss broader ideas on injecting clustering results into course design.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA
©2019 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287459

ACM Reference format
David A. Joyner, Ryan Arrison, Mehnaz Ruksana, Evi Salguero, Zida Wang, Ben Wellington, and Kevin Yin. 2019. From Clusters to Content: Using Code Clustering for Course Improvement. In *Proceedings of 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287459

1 Introduction

One characteristic of large computer science courses is the presence of numerous different solutions to coding problems posed on homework assignments and tests. These solutions exist at an interesting intersection of more mathematical problems and more freeform problems: like mathematical problems, there exist objective criteria for correctness, but like essays or short-answers, there is a nearly infinite number of ways to approach the problems correctly. Many of these different approaches may have superficial differences like differing variable names, but deeper similarity in their underlying structure and behavior [9].

CS educators need to be able to understand the breadth of student answers to inform their own teaching and to deliver appropriate feedback to the right students. However, manually coding student submissions for the type of approach taken is an arduous process, and the work involved grows linearly with scale. Efforts in other areas have been made to cluster student solutions for more efficient feedback and evaluation [1][15][22], but CS education has a unique opportunity due to the simulatability and underlying objective similarity between students' answers.

To address this, multiple initiatives have arisen to perform this processing automatically (e.g. [5]). Most prior research, however, has focused on the mechanics of these systems rather than their use as a tool for researching patterns of student submissions and using those patterns to inform content revisions, tailored feedback, or new initiatives. In this research, we put one such system—Sense, the eponymous name of the company that developed it—to work in an online CS1 class. We begin by giving a brief background of the course in which the tool is used, and then provide a deeper look at some of the results that were uncovered through this analysis. We do not claim that these results reflect general trends in CS1 as we can only generalize to our instruction and delivery style, but rather they are emblematic of the kinds of analysis and results that can be generated using clustering tools such as Sense. Then, we discuss the immediate revisions made and planned for the course based on these results, as well as some more ambitious proposals for integrating content clustering more directly into the student-facing course experience. This analysis and results together demonstrate the potential power of using code clustering tools for course content improvement.

2 Related Work

Significant work has been performed on clustering student responses based on similarity, both in computer science education and in education more broadly. Tools like GradeScope [22], Powergrading [1], and Mathematical Language Processing [13] address this issue in short answer or open math problems by first clustering student submissions together, then mapping those to tailored feedback. GradeScope in particular uses these clusters for emergent rubrics where instructors can modify point deductions for individual errors and have those modifications propagate to all student assignments present in similar clusters. More commonly, work has also been done for constructing expert systems with in-built knowledge of common error types [10][11].

More specifically within computer science education, OverCode [5] and other tools [18][19] address the same need by clustering and visualizing student responses to coding problems. Other work has been performed to generate formative feedback for students based on error models [11][12][24], match prewritten code feedback to new submissions [7][14][17], or identify the differences between new submissions and working submissions for tailored feedback [6][16][21][25][26][27].

Similar work has been done with different goals as well. Much of the early work on clustering student code submissions used abstract syntax trees to evaluate the likelihood that code plagiarism had taken place [20], a technique that has since been extended to educational environments [2][8].

Notably, however, despite the massive number of initiatives in this area, the significant focus has been on delivering individual feedback to the student. While this is without question a valuable goal, it is not the only relevant application of these technologies. Rather than relying on the system to deliver content feedback directly to students, it is also possible that systems like these could instead give instructors information about student patterns of interaction to be used in modifying the content itself.

3 Course Background

This analysis took place as part of the delivery of an online CS1 class. The class is simultaneously offered as a for-credit class at a major public research university in the United States and as a set of Massive Open Online Courses (MOOCs) to the public on the edX platform. Since its inception in January 2017, 785 students have taken the class for credit, and nearly 1,673 have completed a public offering of the course as a MOOC. The class teaches Python 3 and presupposes no prior CS knowledge; it is intended as the first class that new CS majors take and also fulfills the CS requirement for all majors at the university.

The analysis here more narrowly targeted for-credit students' responses from two consecutive semesters (approximately 350 total students) as the goal was to specifically revise the course content for the patterns present among these for-credit students.

As part of the course experience, students complete approximately 350 programming problems. These problems come in three forms: exercises, which are interspersed between short video lectures; problems, which are collected into problem sets at the end of each course chapter; and tests, which are delivered as part of timed and digitally proctored assessments. In all three types of problems, students use a web-based programming interface that generates immediate feedback and evaluation on the correctness of their responses. A limited hint generation system is built in based on an expert system monitoring for the occurrence of certain code or output errors, but most evaluation takes place based on a comparison of student code output to the output of a correct answer to the same prompt.

4 Clustering Results

For this analysis, we used Sense to cluster student responses to every problem on each of six of the course's problem sets. Each problem set is topic-specific: these six sets covered functions, error handling, strings, lists, file I/O, and dictionaries. Prior to these problem sets, students learn loops, conditionals, and operators, and so those concepts exist in their responses to these questions as well. It is important to note that the goal of this work is not to demonstrate a list of common errors and misconceptions in CS1 in general, but rather to show how clustering analyses can be used to uncover such patterns within specific courses with minimal effort.

4.1 Clustering Example: The Rainfall Problem

The sections below cover the high-level trends uncovered in each problem set, but to understand these results, it is useful to first look more deeply at the analysis of a single problem. One problem outside these problem sets that we evaluated was a variation on the famous [4] Rainfall problem [23], which was offered during the third timed and proctored test in the course. Due to technical constraints, rather than have students repeatedly get another piece of user input, our variation supplied students with a list of observed rainfall values as integers, into

which -1 was inserted somewhere after the first digit. Students were asked to average and return all integers before -1. Most students answered the problem correctly (184 out of 190 who attempted it in one semester), although we questioned whether the modifications made to read from a list instead of from user input fundamentally changed the difficulty of the problem.

To answer this question, we uploaded student submissions to Sense. We found first of all that the vast majority of students opted to use a for loop: only 4% of all student submissions used a while loop, which is the method that would more similarly match the traditional framing of the Rainfall problem. 67% of the submissions used a for loop, but broke the loop (either by calling `break` or by returning from within the loop) once it encountered a -1. So, while only 4% of students used a loop definition that would work for the traditional rainfall problem, 71% had a loop body that would generally work for the traditional problem.

Among the remaining 29% of responses, we observed some additional interesting variation. 8% of submissions first modified the list to remove all values after -1, either with the `del` command or by slicing the list until the first instance of -1, found through use of the `list.find()` or `list.index()` methods. They then averaged the remaining list, some with a for loop and some by calling `sum(list)/len(list)`. Interestingly, the `sum` function is not covered in the course material. An additional 8% of students took a similar approach, but without using the `find()` or `index()` method. They initially looped over the list to find the index of -1, then either sliced the list or looped over the list again until that index. While not common, this reflected to us the need to reemphasize list slicing and indexing, which previously had been relegated primarily to a callback to the chapter on strings.

Among the remaining 13% of responses, some entirely unique observations were observed; some were particularly sophisticated, while some were particularly suboptimal. On the sophisticated side, one student wrote two functions: one to average any list, and one to call that function on the list sliced up until the index of -1; another student completed the problem in a single line. These sophisticated responses were fruitful to share with students as exemplary answers. On the suboptimal side, one student generated an entirely new list and added values to it until -1 was reached; another student used nested for-loops to re-find the index of -1 for every value in the list.

This analysis was performed for every one of the 70 problems across the six problem sets. The sections below summarize the interesting trends observed across the problems that we analyzed.

4.2 Functions

The 11th chapter of the course covered functions. For this chapter, the problem set contained twelve problems. Evaluating these problems, we discovered three primary trends.

First, we found that most students' submissions contain some unnecessary structural component, such as an else statement in which nothing really happens like adding zero to a count. For example, one problem asked students to check whether or not a string contained a series of three sevens. We found that 7% of students created an else block in which they either added zero to their count or used the keyword continue. This also occurred in another problem which asked students to find the average word length from a string. 26% of students used pass or "reassigned" a variable without changing its value, or added zero to their count.

Second, we found that when using a formula, students often do their calculations in one or two long lines rather than breaking it up into parts. For example, one problem asked students to calculate the damage a Pokemon would face based on various parameters such as type and level. In this problem, though a lengthy formula was involved, only 27% of students broke their calculations into several parts. Another problem asked students to determine whether a given year was a leap year. We found that 17% of students did their calculations all in one line.

Third, we found that students would sometimes use brute force rather than built in functions or functions of their own. For example, one problem asked the students to make a countdown given the range of numbers. Instead of using Python's range function and a loop, 11% of students manually printed each number in the countdown. In another problem, students were asked to find the average word length given a string. This problem required the students to make three functions, one for word count, one for letter count, and one for average word length. To calculate letter count, students could use their word count function since this accounts for the number of characters that are not letters (spaces) but only 5% did while others recalculated the number of spaces.

4.3 Error Handling

The 12th chapter of the course covered error handling. For this chapter, the problem set contained eight coding problems. Evaluating these problems, we found two primary trends.

The first trend is that as exercises rose in complexity, students demonstrated a stronger tendency to store the result of an expression into a variable prior to printing it, even if printing it was the only remaining task to be performed with the result of the expression. One early problem asked students to attempt up to three calculations, the first two of which could cause an error. 48% of students printed the expressions' results directly, while 43% attempted the expression, stored its result in a variable, and then printed it if an error had not arisen.

Two problems later, students were asked to find pressure given other values using the expression PV = nRT. Here, 71% of students stored the result of an expression in a variable, then later returned it. Only 22% returned the result directly, even though this approach led to fewer lines of code and a more straightforward algorithm.

Secondly, in this chapter, students were able to individually formulate their own algorithms and approaches to more challenging coding prompts. In one later problem, students were asked to write a word count function that could correctly ignore consecutive spaces. 42% of students used a "forward-checking" algorithm, where when encountering a space, the function would check the next character in the string before incrementing

the word count. 18% instead implemented a "backwards-check" algorithm with a boolean variable that tracked whether the previous character was a space. The following problem expanded on this prompt and saw even more significant differentiation: 57 clusters of unique solutions were observed in the next problem, which removed the assumption that the string would start with a word and required students to take special steps in the event of strings with all punctuation marks or no characters.

4.4 Strings

The 14th chapter of the course covers strings, including indexing, splicing, and their context as a data structure. This chapter included a total of fourteen coding problems, the first eight intended for all students and the last six handling more advanced concepts. Two of the most prominent patterns included the use of the `.find()` method as opposed to a range for-loop and a common tendency for the `len()` function to cause an index error, and misunderstanding the difference between strings and lists of strings.

First, as students gain more and more experience using Python and for-loops, they ought to realize that using `for i in range()` is advantageous because each iteration grants both the index and the piece of data through indexing brackets. However, many students in this chapter opted to use a for-each loop on their strings, utilizing the `.find()` method to retrieve the index of their values. This approach is suboptimal, however: even when it works, it is more complex and computationally intensive, and it also fails whenever the string has the same character multiple times.

Second, it is common in Python to use the length of a string as a parameter in the `range()` function, such as `for i in range(len(string))`. In this case, the range begins at zero and ends with `len(string)-1`, as upper bounds are exclusive. However, that upper bound is invisible to students; thus, when they attempt to use the upper bound outside of a loop, they tend to use `len(string)`. When Python encounters this, it will throw an index error because the loop tries to iterate past the length of the string.

Third, the chapter teaches students the `split()` method, which splits a string into a list of strings; however, students do not cover lists until the next chapter. Thus, there are several instances where students treat the output of splitting a string as a string itself, such as trying to call `upper()` on the entire list.

4.5 Lists

The 15th chapter of the course introduced the topic of lists in Python. For this chapter, the problem set contained seven coding problems and four advanced coding problems. Evaluating these problems, we discovered three primary trends.

The first major trend is that students very much preferred to use if statements vs if-else statements. In one question, students were asked to find names that appeared in one list but not another. Most (78%) students created an empty list, then ran a for loop through the first list of names with a nested if-conditional that checked if the name in the first list was not in second. If that was true, they append that name to the empty list, and return that list at the end of the function. A smaller subset of students (10%) used the same process, with the exception being the nested if-conditional. They instead used an if-else conditional, where the if-statement determined if the name on the first list was in the second list. If it was, then pass; else, append that name to the empty list. This method was far less common and the code was longer than it could be, but it remained valid. Notably, this countered an earlier trend where students included structurally unnecessary elements in their code; it appears that as the course moves on, students develop more of a comfort with this abbreviated representation and fewer include unnecessary components. Similarly, in another question, students were asked to write a function that takes in a a list of integers and to find the average of all the integers greater than 100. The most common method (63%) was to create a variable and a counter both equal to zero, then loop through the list. A nested if-conditional would then check if the integer was greater than 100. If it was, then it was added to the sum and 1 was added to the counter; at the end, the average was returned. As before, there was a small group of students (21%) who used an `else` as well, typically with `pass` or `continue` as the only body of the else block.

The second trend is that students tended to understand, and even over-rely, on some list functions like indexing, appending, sorting, and iteration. In one problem, students were asked to find the movie with the highest sales from a list of tuples, each of which contained the movie title and the total ticket sales. Most students (51%) used a for loop to implement a typical linear search, looping through each tuple and analyzing the second index of the tuple. From there they found the maximum sale and returned the movie name. A smaller but significant group of students (19%) created an empty list, added all the sales to the new list, and then used the `sort()` method or `max()` function to get the highest gross. They then used another linear search to find which movie corresponded to that sale. In another question, students were given a list that contained integers and asked to find if three sevens in a row appeared. The largest group of students (34%) looped through the list by position, which allowed them to check the value at the corresponding index in the list and the following two values. However, the next largest group instead used a more complex reasoning structure to count the number of consecutive 7s encountered so far and reset it when a non-7 digit was encountered. This mirrored the preference for using for loops without the range function noted in the analysis of strings. Still others, interestingly, joined the list into a string and searched for "777" or took other more innovative approaches.

The third trend is that students struggled to use efficient code, particularly regarding the return in functions. In one question, students were asked to define a function which took two parameters, a list of answers, and a list of the answer key, and to compare the answers to the key. However, there was the possibility that the answer list and the key list were of different lengths; students were instructed to return -1 in that instance. Students did so correctly, but many set a variable equal to -1

then returned the variable, mirroring in part the pbservation earlier regarding a reluctance of some students to return an expression's result directly. Others even defined a variable equal to the string "-1", then converted the string to integer, then returned the integer. In another question, students were asked to write a function that found a triangle's hypotenuse and angle based on three parameters, the opposite and adjacent lengths of the triangle and a boolean that determined whether to use degrees or radians. Many students did so efficiently, but some took a longer approach, where they would set variables equal to what they wanted to return, then returned those variables.

4.6 File Input/Output

The 16th chapter of the course covered File Input and Output. For this chapter, the problem set contained ten problems. Evaluating these problems, we discovered two interesting trends.

First, while the course material covers a few different ways to read files, students appear to prefer to iterate over the open file. On a problem that requires students to simply read lines from a file and store them as tuples, 63% chose to use `for line in file` instead of methods like `read()` and `readlines()`. While there is nothing inherently wrong with that approach for this problem, it is somewhat more limiting in what can be done with it, and this limiting factor is seen arising in later problems where students need to re-read the file more than once. Upon reaching these problems, most students stick to this approach even though it demands more convoluted logic, like closing and reopening the file.

Second, regarding closing files, most students closed files at the end of the function rather than as soon as file-reading had completed. One problem, for example, had 82% of students closing a file as the last line of code, even though they only used the file at the very beginning of the code. For many of these students, closing the file actually took place *after* the return statement, meaning that the file was never actually closed.

4.7 Dictionaries

The 17th chapter of the course covered dictionaries. This chapter had twelve coding problems and three advanced coding problems in the problem set. Three trends stuck out to us in this chapter: the use of for loops and nested for loops in place of populating a dictionary more directly, the use of lists to avoid using dictionaries directly, and the use of try and except.

While most students took understandable approaches, the first trend we observed was that a non-trivial number of students took one of several approaches that effectively bypassed using dictionaries as they were constructed to be used. For example, one problem asked students to write a function to take a string and return an integer:list dictionary, where the keys were word lengths and the values were lists of words with the corresponding length. Some students took approaches like iterating through the keys to find the right key before appending to the list rather than just looking up the key directly.

Second, some students—presumably more comfortable working with lists from the previous chapter—unpacked dictionaries into lists for processing. These students appeared to believe that dictionaries were effectively two disjoined lists rather than understanding that keys actually point to values in the dictionary. These students thus did not grasp that an easier way to populate a dictionary would have been to use the key to modify the values, and instead they would create a list and then use indexing to make a key variable. Often, they would then use a two-dimensional list and nested for loops for the values.

Third, we found an interesting carry over from the chapter on error handling. On the one hand, we found a non-trivial number of students who, rather than handling edge cases deliberately, assumed the code would operate correctly and caught errors if they arose. This reflected a mature understanding of the role error handling could play. However, another non-trivial set of students appeared to use try and except blocks more haphazardly. These students included error-handling code that did not actually react to errors; they also wrapped try blocks around code that would not generate errors in the context of the problem. Still others used no error handling code and were forced to handle potential problems more deliberately by, for example, preemptively checking if values were the right types for subsequent operations.

5 Content Modifications

As noted above, most prior research using tools like Sense has focused on delivering students individualized feedback based on their answer patterns. This is a highly desirable goal and not in conflict with our approach; however, we also note that by delivering this feedback only to students who commit the error, we risk excluding students who possess the underlying misconception but never demonstrate it in their work. For example, we noted above that in the chapter on functions, many students have an unnecessary structural component of their code, like a statement merely saying `else: pass`. This suggests either a misconception that every `if` must have an `else`, or an undesirable stylistic preference to have an `else` for every `if`. To resolve this, we authored additional content deliberately exploring whether an `else` is required for every `if`, in order to ensure that a correct understanding is reinforced for every student.

A number of similar modifications have either been made or are planned for the near future based specifically on the outcomes of this analysis. These include:

- Authoring exercises on the readability of long formulas.
- Authoring exercises reinforcing the ability and desirability of returning the results of simple expressions directly.
- Authoring additional content regarding available functions like `range` and `sum`, and embedded reminders on appropriate times to use these functions.
- Authoring additional instructional material on the situations under which using `string.find()` or `string.index()` will fail, and alternatives to avoid needing those method calls in the first place.
- Moving content requiring students to process the result of calling `string.split()` into the chapter on lists.

- Revising the autograder to more intelligently assess whether a file has been closed.
- Authoring additional content regarding the scope of file reading and the benefits of closing a file as soon as its contents have been read into the program.
- Author additional content regarding the scope of variables in a function, specifically targeting inadvertently accessing variables from outside the function rather than the function's own parameters.
- Authoring additional content regarding the usefulness of dictionaries rather than just syntax for using dictionaries.
- Authoring content guiding students on how to understand whether error handling code is necessary rather than merely the structure of error handling code.

6 Conclusion

Significant work has been done, both in CS education and in education more broadly, on clustering student answers to support individualized feedback. However, most of this research has stopped there in its application of these clustering techniques. Once clustered, however, there are significant additional improvements that can be made to course content based on the knowledge gained from this analysis.

In this work, we have demonstrated one such approach to using the results of a sophisticated code-clustering algorithm to inform the revision of the course content itself. We uploaded over 14,000 student code submissions to the clustering system spanning 70 different exercises. We then used the results to inform general revisions to the content to specifically target the common misconceptions and errors observed in the results of that analysis.

We do not claim that this approach is generally superior or preferable to using clustering for tailored feedback, but we do argue that using clustering to inform more broad content revision has additional benefits. Modifying the course content itself removes the reliance that every student demonstrates every misconception they hold in their work. Tailored feedback is excellent for giving individualized and just-in-time correction, but it may miss students who for one reason or another never demonstrate the need for that correction. Revising the material more broadly targets them as well, and it reinforces the correct conception among students who may not have misconceptions but are not firm in their correct conceptions.

We have not performed a systematic analysis of the change in student behavior based on these changes, but anecdotally we have noticed significant changes. One prominent change occurred in response to new content targeting variable scope and functions: we noted a significant drop in the number of questions from students who did not realize that their functions were looking at variables defined outside the function rather than variables passed in through the parameter list. This reflected to us the value in targeting content specifically to the misconceptions and errors observed through clustering analyses like these.

6.1 Limitations

The results from our specific analyses listed in this research likely are not generalizable out to other classes as well. Student patterns in our class are driven by our own instructional material, and different approaches will lead to different patterns. This research is a demonstration of the use of code clustering in rather than a list of common errors and misconceptions in CS1.

That said, the technology used in these analyses may be used for research on common errors. If a shared set of problems is distributed and used across multiple classes, then a clustering analysis may identify what errors are specific to individual instructional styles and what errors are more inherent to the content. Such a shared corpus of exercises would be valuable in assessing what is inherently difficult about programming.

6.2 Future Work

Others have looked at using clustering for tailoring feedback or detecting plagiarism, and we have looked at using clustering for revising content. There are significant other potential applications of submission clustering that we plan to explore as well. First, this clustering may also serve as a valuable research tool, both for broader CS education research and in evaluating other improvements made to the class. The changes documented here targeted specific desired changes in the patterns of students' answers on the problem sets. Follow-up analyses of future semesters' submissions may allow us to establish the effectiveness of those changes.

Second, on our preliminary research, we have observed that there are some clusters of students that appear to group together across multiple assignments. This is unsurprising on those problems where the majority of students tend to take the same approach, but it is curious to note instances where the same small group of students takes the same unorthodox approach across multiple exercises. Are they a study group who happens to arrive at the same ideas together? Are they a group of people with a particular background that causes them to favor some answers over others? Are they a circle of people copying answers from one another? The usage of clustering to track groups of people across problems has significant potential.

Third, clustering may also be used to foster greater social connectedness in a class, an approach that could be particularly relevant for students in an asynchronous online class or MOOC. For example, one can imagine that upon submitting an answer, a clustering system finds the similar answers from previous submissions and alerts the student, "Your approach to the problem is similar to 72% of prior individuals! Would you like to see some of the approaches the other 28% take?" In addition to some of the benefits of tailoring the selection of alternate examples to the individual's own approach, this may also help the student contextualize their own meta-knowledge and establish if they are adopting common behaviors.

REFERENCES

[1] Sumit Basu, Chuck Jacobs & Lucy Vanderwende (2013). Powergrading: a clustering approach to amplify human effort for short answer grading. *Transactions of the Association for Computational Linguistics, 1*, 391-402.

[2] Sébastien Combéfis & Arnaud Schils (2016, November). Automatic programming error class identification with code plagiarism-based clustering. In *Proceedings of the 2nd International Code Hunt Workshop on Educational Software Engineering*, 1-6. ACM.

[3] Molly Q. Feldman, Ji Yong Cho, Monica Ong, Sumit Gulwani, Zoran Popović, and Erik Andersen. Automatic Diagnosis of Students' Misconceptions in K-8 Mathematics. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 264. ACM.

[4] Kathi Fisler. (2014, July). The recurring rainfall problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, 35-42. ACM.

[5] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. (2015). OverCode: Visualizing variation in student solutions to programming problems at scale. ACM Transactions on Computer-Human Interaction (TOCHI), 22(2), 7.

[6] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. (2018, June). Automated clustering and program repair for introductory programming assignments. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 465-480). ACM.

[7] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. (2017, April). Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, 89-98. ACM.

[8] David Hovemeyer, Arto Hellas, Andrew Petersen, and Jaime Spacco. (2016, August). Control-flow-only abstract syntax trees for analyzing students' programming progress. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 63-72. ACM.

[9] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. (2013, June). Syntactic and functional variability of a million code submissions in a machine learning MOOC. In *AIED 2013 Workshops Proceedings Volume*, 25.

[10] Magdalena Jankowska, Colin Conrad, Jabez Harris, and Vlado Kešelj. (2018). N-Gram Based Approach for Automatic Prediction of Essay Rubric Marks. In *Advances in Artificial Intelligence: 31st Canadian Conference on Artificial Intelligence, Canadian AI 2018, Toronto, ON, Canada, May 8–11, 2018, Proceedings 31*, 298-303. Springer International Publishing.

[11] David A. Joyner. (2018, June). Intelligent Evaluation and Feedback in Support of a Credit-Bearing MOOC. In *Proceedings of the 19th International Conference on Artificial Intelligence in Education,* 166-170. Springer, Cham.

[12] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. (2016, November). Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 739-750. ACM.

[13] Andrew S. Lan, Divyanshu Vats, Andrew E. Waters, and Richard G. Baraniuk. (2015, March). Mathematical language processing: Automatic grading and feedback for open response mathematical questions. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, 167-176. ACM.

[14] Victor J. Marin, Tobin Pereira, Srinivas Sridharan, and Carlos R. Rivero. (2017, April). Automated personalized feedback in introductory Java programming MOOCs. In *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 1259-1270. IEEE.

[15] Agathe Merceron and Kalina Yacef. (2005). Clustering students to help evaluate learning. In *Technology Enhanced Learning*, 31-42. Springer, Boston, MA.

[16] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. (2014, April). Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web*, 491-502. ACM.

[17] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. (2017, June). Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 92-97. ACM.

[18] Ronen Tal-Botzer (2018). U.S. Patent Application No. 15/549,179.

[19] Lina F. Rosales-Castro, Laura A. Chaparro-Gutiérrez, Andrés F. Cruz-Salinas, Felipe Restrepo-Calle, Jorge Camargo, and Fabio A. González. (2016, November). An Interactive Tool to Support Student Assessment in Programming Assignments. In *Ibero-American Conference on Artificial Intelligence*, 404-414. Springer, Cham.

[20] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. (2003, June). Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of Data*, 76-85. ACM.

[21] Saksham Sharma, Pallav Agarwal, Parv Mor, and Amey Karkare. (2018, June). TipsC: tips and corrections for programming MOOCs. In *Proceedings of the 19th International Conference on Artificial Intelligence in Education*, 322-326. Springer, Cham.

[22] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. (2017, April). Gradescope: a fast, flexible, and fair system for scalable assessment of handwritten work. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, 81-88. ACM.

[23] Eliot Soloway. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29*(9), 850-858.

[24] Kristin Stephens-Martinez, An Ju, Krishna Parashar, Regina Ongowarsito, Nikunj Jain, Sreesha Venkat, and Armando Fox. (2017, August). Taking Advantage of Scale by Analyzing Frequent Constructed-Response, Code Tracing Wrong Answers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, 56-64. ACM.

[25] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. (2017, October). TraceDiff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 107-115. IEEE.

[26] Ke Wang, Rishabh Singh, and Zhendong Su. (2018, June). Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 481-495. ACM.

[27] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. (2017, August). A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 740-751. ACM.