# Evaluating CodeClusters for Effectively Providing Feedback on Code Submissions

Teemu Koivisto
*Department of Computer Science*
*Aalto University*
Espoo, Finland
teemu.koivisto@alumni.helsinki.fi

Arto Hellas
*Department of Computer Science*
*Aalto University*
Espoo, Finland
arto.hellas@aalto.fi

***Abstract*—*Full research paper*—Most introductory programming courses rely on the use of automated assessment for grading programming assignments. While such systems save teachers' time by eliminating manual grading, the submissions may not be reviewed by the teachers at all, losing valuable insight into how students solve the assignments. In this paper, we introduce CodeClusters which provides teachers' a quick overview of general patterns in code submissions. The main features of the system are full-text search and N-gram -based similarity detection model that can cluster and subset the code by various aspects such as AST similarity or software also has It has also an interface for streamlining the writing of feedback to multiple submissions at once. CodeClusters has been primarily designed to be used jointly with automated assessment systems where automated tests would assess the functionality of the code, and CodeClusters would be used for gaining a higher-level view of programming patterns and for writing feedback to students. CodeClusters was evaluated in a think-aloud study with university lecturers responsible for programming courses at two research-first universities. The lecturers were pleased by the possibility of providing better feedback to students quickly, saw it could improve the quality of their courses over solely automated assessment, and expressed interest in using CodeClusters in their own courses.**

***Index Terms*—source code submissions, source code analysis, clustering, feedback**

## I. Introduction

One of the key components in many introductory programming courses is hands-on practice with programming assignments that help students learn both syntax and semantics. A key part of this hands-on practice is feedback, which helps students to adjust their work to better match the expectations [1]–[3]. As an introductory programming course may receive hundreds or thousands of programming assignment submissions each week [4]–[7], writing feedback manually for each student is often not feasible.

A common solution for providing large quantities of feedback is the use of automated assessment systems, which run a set of automated tests on programming assignment submissions and provide feedback on the correctness of the submissions [8]–[12]. The use of automated assessment allows a faster feedback cycle when compared to manual reviewing, and is effectively available around the clock [12]–[14]. One of the benefits is also consistency as the same grading criteria is applied to every student submission, thus providing same grade and feedback on similar programming errors [12]–[14].

The use of automated assessment, however, involves a trade-off. While the use of automated assessment removes the need to manually assess submitted assignments, teachers can become dependent on the scores from the automated assessment systems, and may not be as involved with manual inspection of students' submissions as they would without such systems. This in turn may lead to teachers becoming unsure about the problems that students face, which makes it harder to design classroom interventions to address gaps in students' understanding.

In this work, we introduce and evaluate CodeClusters, which has been designed for effective manual reviewing of programming assignment submissions as well as for writing feedback for multiple submissions and aspects of submissions at once. Similar prior art to the system are OverCode [4], Codewebs [5], and Divide and Correct [15]. In OverCode stacks of canonicalized source code lines are used to cluster the submissions, Codewebs enables searching and clustering the code by similar code phrases, and Divide and Correct clustering using TF-IDF similarities with k-medoids algorithm. CodeClusters uses N-grams built from tokenized abstract syntax trees for clustering with various configurable clustering algorithms. It also provides search functionality that can use either software metrics or string-based matching. Additionally, teachers can create review flows to automate their feedback with the parameters they have found suitable. Of the related systems, Divide and Correct appears closest although it has been designed for natural language submissions and uses only a fixed approach for clustering [15].

This article is organized as follows. Subsequently, in Section II, we discuss the analysis of source code and programming assignment submissions. In Section III, we present CodeClusters. Section IV outlines the evaluation methodology of CodeClusters, including the structure of the interviews, and Section V outlines and discusses the results of the evaluation. Finally, Section VI summarizes the work and outlines future research directions.

## II. Background

Here, we outline the background for the development of the CodeClusters. We first discuss reviewing students' code, which is followed by an outline of ways for representing code for automated analysis. Finally, we briefly discuss source code similarity detection.

### A. Reviewing code submissions

Student code has been reviewed in multiple ways to study learning [16], misconceptions [17], likely grades [18] as well as code quality [12], [16] and complexity [12]. The approaches of reviewing student assignments can be broadly categorized as either manual or automatic.

Manual review depends typically heavily on teachers and teaching assistants (TAs). One of the benefits of manual reviews is gaining a view of the types of problems that students struggle with; the act of being a TA, as an example, can be a valuable experience by itself [19]. Providing consistent reviews can, however, be difficult. As an example, when seeking to build an automated feedback system out of previously given reviews, Rogers et al. [20] noted that one of the key problems was inconsistency in grades and feedback from TAs. Similarly, when asking TAs to cluster code submissions, Glassman et al. [21] observed that TAs often overlooked presentational details and preferred structural similarities, which led to somewhat distinct clusters between TAs.

Indeed, there are multiple levels of variations between submissions. Luxton-Reilly et al. [22] suggested that these variations can be categorized into three hierarchical levels. At the first level, the control-flow structure defines the main aspects of the program. At the second level, if two submissions share the same control-flow structure, variations in program syntax that do not contribute to control-flow can be analyzed. At the third and final level, if the syntax of the submissions also matches, presentational variations such as the differences in formatting and exact tokens and their order can be studied [22].

Programmatic reviewing of code can be further divided into two categories; static and dynamic analysis, where static analysis tools study the code without executing the code, while dynamic analysis can also execute the code [23]. A popular type of static analysis tool is a program called "linter" [24], which provides feedback on code structure and formatting. The use of linters can help developers maintain a uniform style, reduce maintenance, enhance readability, and avoid errors in their code [24]. Although linters have been used in programming courses [16], [25], Crow et al. code quality aspects are not discussed in some introductory programming courses [26]. Automated tests that are run on student submissions for the purposes of automated assessment [10] can be seen as a form of dynamic analysis, and one can also write in guiding messages or "scaffolding" into the automated tests [27]; there exists also the possibility of identifying transformations that are needed for the submission to match a correct solution, as suggested e.g. in [28]–[30].

### B. Code Representations and Metrics

Code can be represented in multiple ways. In general, representations are built by first parsing the code into a stream of tokens. The tokens are then parsed into a parse tree, which effectively contains a derivation of the programming language grammar. As parse trees can become relatively large, they are often transformed into abstract syntax trees (ASTs) [31], [32], which offer the full syntactical structure of the program in a hierarchical form.

ASTs can further be standardized and pruned depending on their use. As an example, MistakeBrowser [28] and Codewebs [5] standardize ASTs by removing less important tokens, which can be seen similar to removing stop-words in natural language processing. Then, similar tokens are transformed into a single representation, such as changing `for` and `while` tokens into `loop` tokens. Another option is to focus on the possible execution paths of a program, which is typically modeled as a control-flow graph (CFG), which is effectively a directed cyclic graph of all the possible execution paths a program may take during its execution [31]–[33].

Control-flow graphs remove many of the noisy details of formatting and syntax while retaining the actual behavior of the program. In general, CFGs have also been used to study variations between student submissions [21], [22], [34]; Control-Flow Abstract Syntax Tree (CFAST) [34], for example, models the control structure of the code by pruning ASTs to only contain the tokens that contribute to the control flow of the program. Losing the tokens, however, also implies losing the ability to detect syntactical similarities. This would decrease the overall applicability of models that use only CFGs.

Another possibility is to transform the AST into a sequence of tokens, which is then used to represent the program or parts of the program. Some systems, such as JPlag [35], add separate tokens to the start and end of statements, such as "BEGIN_IF" and "END_IF", to help detect nested structures in the data. These tokens are often used as a set of sequences (N-grams), each sequence with a number of tokens [36], where the set of token sequences is then used to represent the code.

Perhaps unsurprisingly, different code representations and transformations have been also used in static and dynamic analysis. As an example, the Cyclomatic complexity metric uses CFGs in its calculation of complexity [37], while the Assignments, Branches, Conditionals (ABC) metric counts the specific structural parts of the program [38].

### C. Similarity Detection

Much of the research in similarity detection of program code has focused on the detection of plagiarism, software license violations, and clones. Code similarity can be broadly categorized into three levels: purpose, algorithm, and implementation level [39]. If two programs both sort numbers by ascending order, they can be considered similar at the purpose level. If they share the same algorithm, at the algorithm level, and if their implementation of the algorithm is the same, at the implementation level. Due to the difficulty of deciding whether

two programs are equal at the purpose or algorithm level, most similarity methods work only at the implementation level [40] – if we expect that two students have solved the same exercise, we can already assume that the programs that they have written are the same at the purpose level.

In addition to the above examples, similarity detection has been also used for providing feedback to students [1], [4]–[7], [15], [25] using clustered solutions so that one can, in principle, provide one feedback that then is propagated to the whole cluster. Systems such as OverCode, Codewebs, and Divide and Correct use similarity detection, and their results are used to cluster the submissions. Many of the systems use derivations of ASTs as their representation, but the used similarity measures vary. OverCode uses a custom algorithm that matches canonicalized stacks together, Codewebs uses probabilistic semantical equivalence to find the most used context for given AST subtree, and Divide and Correct uses multiple feature vectors which are compared using cosine similarity [4], [5], [15].

Deciding how the similarities are calculated is crucial in producing accurate results, yet as discussed in Section II-A, there might be multiple similarity levels and their measurement would be susceptible to subjective biases by human reviewers. When considering the actual clustering algorithms for finding similar source codes, there are a wide variety of algorithms such as k-means, DBSCAN and OPTICS (see e.g. [41]–[44]), which have also been used in previous studies with students' submitted source code [7], [15], [21], [25].

## III. CodeClusters

As a general description, CodeClusters[1] is a web application that is used to study code submissions and to provide feedback on the submissions. CodeClusters includes search and modeling functionality, as well as an easy interface for creating feedback to groups of students. The present version supports Java program code submissions and instead of a one-size-fits-all model, it is designed to allow the composition of different parts of a review process: searching, modeling, and providing feedback, to provide teachers greater flexibility in automating their review process. The high-level view of how CodeClusters has been designed to be used is shown in Figure 1.

### A. Implementation

At its core, CodeClusters is a web application that can be accessed a web browser. It is divided into 6 parts that are run as individual Docker containers. The containers are managed by Docker Compose and the server configuration is written as `docker-compose.yaml` files. This simplifies the development and deployment of the system and using containers allows managing the various dependencies of the servers easily. While CodeClusters provides a single-page web application (SPA) front-end written using React that is used for

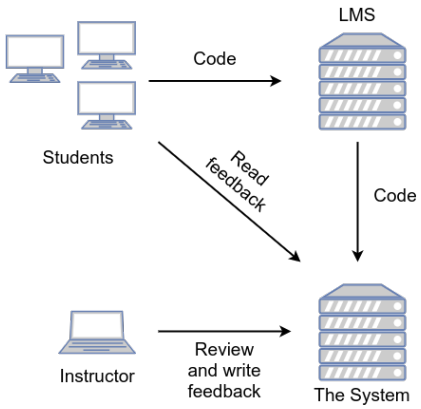


Fig. 1. An overview of the flow of using CodeClusters. Students submit their code to an LMS, which then sends the code to CodeClusters. Teachers use CodeClusters to review the submissions and to provide written feedback to them, which is then read by students.

interacting with the parts of the application, here we discuss the key parts of the system that allows the review process.

*1) Database:* CodeClusters uses PostgreSQL as it is widely accepted in the industry as the leading open-source relational database. The database is accessed mainly through the application server, but the modeling server is also provided access to avoid having to pass possible large quantities of data from the application server.

*2) Search server:* The search functionality has been implemented using Apache Solr[2], which is based on the Apache Lucene search engine. The code features indexed into Solr include the code and general metadata of the submissions such as student id and timestamp. The code is tokenized by line breaks and whitespace, which was observed to offer the most flexibility in searching various special characters and exact patterns compared to pruning the code, despite possible issues with improperly formatted code. Other indexed features are software metrics, generated by the Checkstyle[3] static analysis library. In addition, the total sums of various Java keywords are also indexed.

All of the features are searchable from the UI by either string queries that support Lucene operators, filters that specify ranges of desired values or facets, a Lucene-specific feature that enables subsetting the data based on indexed values. The configuration of Solr was relatively small. For ranking the documents Solr uses Okapi BM25, a probabilistic IR model, which at the time of the development was the default IR model for Lucene. Modifying or changing it to another model, for example, TF-IDF, was not deemed necessary in evaluations conducted during the development of CodeClusters.

*3) Modeling server:* The modeling server was implemented using Python and Flask, utilizing the scientific libraries of the Python ecosystem for clustering. In the present state, the system can process Java program code by either parsing it into

[1]Sources available at https://github.com/Aalto-LeTech/CodeClusters and https://github.com/Aalto-LeTech/CodeClustersModeling

[2]https://lucene.apache.org/solr/

[3]https://checkstyle.sourceforge.io/

ASTs using ANTLR[4] parser generator or generating metrics with Checkstyle static analyzer.

The ASTs are parsed as part of the similarity detection, transformed into an N-gram representation of tokens represented in vector space as TF-IDF matrices and cosine similarities. The modeling functionality in the application allows the manual setting of various parameters such as the used set of N-grams, the used token set, clustering algorithm, and the 2-D dimensionality reduction method. Most of the modeling is implemented using the basic scientific computation libraries such as NumPy and scikit-learn, and the available clustering algorithms are k-means, DBSCAN, HDBSCAN, and OPTICS algorithms. To plot the submissions in a two-dimensional scatter plot, a dimensionality reduction method is used with either t-SNE or UMAP algorithm. All of these options are parameterized and choosable by the user.

### B. Main features

The main features of CodeClusters are an easy-to-use code search functionality, where instructors can search code submissions with code, with metric ranges (e.g. lines of code between given two numbers), and with Lucene-specific operators and filters. Searching code leads to a list of submissions that match the search, where the code for each submission is shown to the user. Submissions can be selected from the list, which then allows writing a review for the selected submissions.

When writing submissions for a set of selected submissions, one can also add one or more tags to the review, which can then be selected for reviewing and for providing feedback. CodeClusters also keeps track of the submissions which have already been reviewed and which have not yet been reviewed, and provides an overview page with a matrix-like view that allows fast inspection of current review status. The matrix-like view also provides fast access to submissions that have not yet been reviewed. Reviews need to be accepted on the overview page before they are sent to the students.

In addition to the search functionality, CodeClusters allows clustering submissions. When clustering submissions, the instructor selects model parameters (defaults are provided) and runs the clustering. The clustering results are shown both with histograms that show the sizes of the clusters and with a 2-dimensional visualization. The Figure 2 shows an example of a clustering attempt; model parameters are chosen and clustering has been performed. Two clusters were identified (labels 0 and 1 on the bottom left-hand side); one of them has 13 submissions, while another has 11 submissions. In this case, there are 53 submissions that were not clustered, indicated with the label -1. Clicking on a bar in the histogram opens a review area similar to the one that is used when searching submissions, which allows reviewing and providing feedback to selected submissions.

CodeClusters also provides the functionality for creating review flows, which are effectively pre-defined searches or clusterings with pre-defined review messages. These can be used, for example, for rapid generation of feedback on code submissions that share properties that are easily defined with the search or clustering functionality.

[4]https://www.antlr.org/

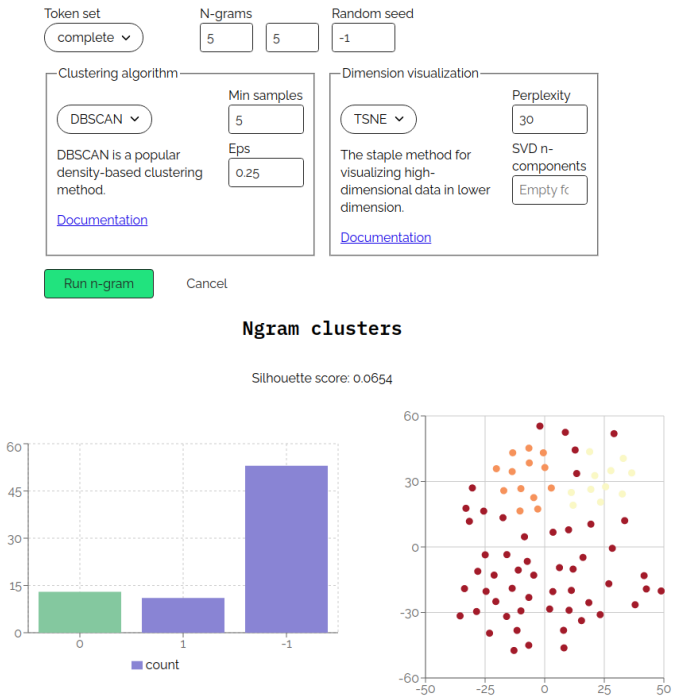


Fig. 2. Sample view in CodeClusters. The teacher has selected an assignment and used CodeClusters to create clusters on the source code submissions. Clicking on a cluster (in the visualizations) leads to the view used for writing feedback.

### C. Related Systems

Here, we briefly discuss three related systems. OverCode [4] and CodeWebs [5] have been designed for providing feedback to large numbers of program submissions at ease, while Divide and Correct [15] works on natural language submissions.

*1) OverCode:* OverCode [4] is a code visualization and exploration tool that helps teachers and TAs to create grading rubrics and to find misconceptions, pedagogical examples, and other interesting patterns. OverCode uses various static and dynamic analysis techniques to reduce the dimensionality of the submissions, including analyzing execution graphs and renaming variables that are used similarly so that they use a common name.

The key feature used in clustering and grading is stacks, which are single lines of canonicalized code that are shown with the number of submissions they occur in. Teachers can select stacks to write feedback about and merge them with other stacks. Unlike systems that use more abstract source code distance metrics, OverCode maintains closeness to the original representations by pattern matching the canonicalized lines of code, making the process straightforward to follow and to understand. On the other hand, as the focus is on stacks, the overall structure of the code is not in focus, which can make it difficult for the teachers to gain a higher-level view of

structural variations in students' solutions. This can also lead to challenges when grading more complex assignments.

*2) Codewebs:* Codewebs [5] is a search engine for programming submissions that helps teachers find submissions that share a similar structure, which can then be used to provide feedback to students; the feedback could cover, for example, misconceptions or other problems. Codewebs uses automated test results and code, both as strings and ASTs, which are indexed for efficient search.

The key features include an inverted index of submissions and the ability to search code with a set of code phrases. These code phrases do not need to be exact; instead, the search engine looks for submissions that are semantically similar to the searched code phrases. Codewebs has been tested on a dataset of million Octave code submissions, and it seems to scale well for grading, studying, and providing feedback to submissions, at least in the context of individual functions.

*3) Divide and Correct:* Divide and Correct [15] is a system designed for reading, grading, and providing feedback to short natural language submissions. It applies similarity detection by combining multiple features and transforming the features into a distance metric, which is then used for clustering the submissions. The resulting clusters can then be graded and provided feedback using a web interface.

The key features of Divide and Correct include hierarchical clustering of submissions, which allows teachers to first divide the submissions into up to 10 clusters, and then further divide the submissions into a maximum of 5 subclusters per cluster. The approach used in the system is based on a previous clustering approach called [45]. While such systems are rarely evaluated by teachers, at least in the articles that describe the systems, Divide and Correct was evaluated in a within-subject study with 25 teachers. The results indicated that while grades were similar in groups that did not use Divide and Correct and groups that used Divide and Correct, teachers provided a lot more feedback when using Divide and Correct. Similarly, using Divide and Correct, teachers were able to go through the submissions roughly three times faster.

## IV. Methodology

We use Design Science Research (DSR) as the basis for our research. DSR "is a research paradigm in which a designer answers questions relevant to human problems via the creation of innovative artifacts, thereby contributing new knowledge to the body of scientific evidence. The designed artifacts are both useful and fundamental in understanding that problem" [46]. Our approach can be defined as exploratory DSR analysis, where the objective is to verify that the problem we outline exists and that the designed artifact has the potential to solve that problem [46], [47].

The starting point for this study was the observation that while professors can use automated assessment tools for grading, reviewing, and providing feedback to the submissions in programming courses is not feasible to a large number of participants. The evaluation itself is conducted as a semi-structured interview complemented with tasks and think-aloud protocol, where we introduced CodeClusters to professors who currently or in the past have been responsible for teaching introductory programming courses.

### A. Structure of the interview

Overall, the interview structure was as follows. First, before the interview, participants answered a preliminary questionnaire, which provided us information about their teaching experience and perceptions related to grading and providing feedback. The interview consisted of two parts, where the participants, after a demo of CodeClusters, solved a set of tasks using CodeClusters. Finally, the participants were given a satisfaction and evaluation survey, where they were asked for their impressions regarding CodeClusters and how they would see its features, the search, modeling, and reviewing, be of use for analyzing and providing feedback to course submissions.

During the interview, the participants were free to voice out their thoughts and to discuss their answers to the tasks at any given point, and to give further justifications for their answers. The interviews were conducted online, where the participant shared their screen as they were solving the problems. Each interview was conducted separately.

### B. Interview tasks and data

The interview tasks were designed to simulate a situation where the participants were reviewing and providing feedback to Java programming course submissions. As the data, we used a total of 110 submissions for the Rainfall Problem – the Rainfall problem is often used in introductory programming courses and considered a relatively difficult problem for novices [48].

The interview tasks were divided into four parts with each part addressing a different aspect of CodeClusters. The parts were ordered; (1) reviewing using searching, (2) reviewing with modeling, (3) creating and using review flows, and (4) reviewing and accepting reviews. To perform the tasks, the teachers logged into CodeClusters that was seeded with a dataset. The tasks can be found in their entirety in the Appendix A.

The first part consisted of searching submissions code using the search server and then reviewing the searched submissions. This required that the participants were able to find the correct controls and to use them to perform various searches. The participants used various features of the Lucene API provided through the search server, such as facets and filters, partially to showcase the search capabilities of CodeClusters.

In the second part, the participants clustered source code submissions with the modeling functionality and then reviewed submissions within the clusters. The participants were not expected to understand the models or their implementations, and necessary parameters for e.g. clustering were provided in the task description. The participants were asked to inspect the generated clusters and to compare them against each other. The objective was to see how well the participants can process the clusters and whether they could find structural patterns amongst them. The purpose of the tasks was to demonstrate

how such models behave in general and what their possible shortcomings are.

In the third part, the participants used the review flow functionality of CodeClusters. Review flows are a feature of CodeClusters that allow the composition of search, modeling, and review into a single executable action that helps automate repetitive parts of the review process. In the task, the participants were asked to create a basic review flow to demonstrate its functionality, including writing a few reviews.

Finally, in the fourth part, the participants were asked to test the review overview functionality, which allows them to see all the submissions in a single view with the provided feedback. They were then asked to edit a few of the associations between the reviews and submissions, and then to accept the reviews, which would lead to sending the reviews to the students.

### C. Analysis of the results

For the analysis, we coded the interviews and combined the results of the coded transcripts of the interviews, the questionnaire answers, and the performance of the participants during the tasks. The questionnaires and the tasks were processed chronologically to create a synthesis of results. As our sample size was small, statistical methods were not used to e.g. look for differences in performance in the participants.

## V. Results and Discussion

### A. Participants and the preliminary questionnaire

In total, five participants from two Universities participated in our study. All participants declared themselves to be university lecturers (similar to Associate Professors in the American system). Many had long careers in academia, which was also voiced in the discussions – one participant reminisced that in the end of the 1980s and at the start of the 1990s, the grading of programming assignments in introductory programming courses was manual, while they had since transferred mostly to automatic grading.

When considering the question "Grading assignments automatically is better than by hand.", the participants voiced that the answer naturally depended on the perspective. Automatic grading was considered as a worse option for students by many, as this often had the consequence of having less personalized feedback. One participant did note, however, that automatic grading does provide the possibility of giving near-instant feedback, which can be more useful to the students, especially when learning the basics of programming. From the instructors' perspective, almost everyone viewed automatic grading better, however, partially due to being heavily constrained by the available time.

The participants answered similarly to the questions "If you had to grade an assignment by hand, what would be your process roughly?" and "If you had to give feedback in writing to student for an assignment, what would be your process roughly?", possibly as the process involved manual writing of feedback. Many also mentioned the use of grading rubrics or checklists that would be used to define and create the criteria for evaluation, including the use of tools such as Rubyric [49]. There was also discussion on the use of Github issues and different communication technologies such as Telegram for providing feedback. In general, the discussions also indicated that grading assignments by hand – especially introductory programming assignments – was very rare.

Every participant agreed that feedback is a very important part of the learning process. At the same time, almost everyone agreed on too few teachers and teaching assistants, alongside having a large number of students, as the main problems with providing feedback to students.

When considering the question "I often check student submissions to see how they have approached a problem.", it was evident that programming submissions were not collectively analyzed, even though some instructors mentioned the possibility of viewing how students construct their solutions using tools such as CodeBrowser [50]. Despite the evidenced lack of manual analysis of programming assignment submissions, all the participants agreed that knowing how students solve programming assignments is very important.

In general, perceptions on the use of automated feedback were mostly positive, although it was mentioned that the quality of automated feedback is another matter.

### B. Interview tasks

The participants were given the task of reviewing a Java programming course submission for outliers or other interesting patterns and to provide feedback based on their findings. All the tasks were scripted and the participants were given help if it became clear that they did not know how to perform a task. This was intentional as asking people to use an unknown system is not without drawbacks, even if the context is familiar.

First, in the search tasks, the participants learned to use the Lucene facets and to select submissions with a cyclomatic complexity of 3. There was only one such submission, intentionally faulty, and when looking at the code, the participants were prompted whether they saw anything wrong with the submission. The submission contained a BIT_AND (&) that was used instead of AND (&&). None of the participants identified this – this could indicate that the participants were either not accustomed to detecting such flaws and the multitude of potentially erroneous ways that one could attempt to solve the assignment.

After a while, all the participants became familiar with the user interface and did not seem to have problems with it. We acknowledge that a longer session with the participants would have been required to verify this conclusively.

Second, in the modeling tasks, the participants learned to divide submissions into clusters with CodeClusters. The participants were asked to run an N-gram model and compare the submissions between the clusters. The model divided the submissions into structurally similar clusters that shared the same syntactical tokens all the way to the individual ordering of the statements. The difference between two clusters might have been, for example, the use of an additional variable or having `else if` instead of `if`.

The comparison proved to be difficult and not many participants were able to detect differences without help. One participant even mentioned that without having submissions side by side, it was not easy to remember what they were comparing against. We noted this as a suggestion for future work.

After the modeling, the participants performed the remaining tasks without considerable problems, including writing feedback to students. In total, the time it took to perform the interview tasks ranged from 30 to 50 minutes, depending on how much the teacher wanted to discuss the system and analyze the submissions.

### C. Satisfaction and system evaluation questionnaire

After the interview tasks, the participants were given a questionnaire that gauged user satisfaction and perceptions of the system.

First of all, when considering the ease of use of CodeClusters, most of the answers were neutral. This was in line with how the participants performed in the tasks and also highlights a need for a good onboarding process. The system also has inherent complexity due to the ability to use the Lucene API and the possibility of setting modeling parameters manually, which likely contributed to the feeling of complexity. Most participants highlighted the unfamiliarity (and unintuitivity) of the UI as the main problems of CodeClusters.

Regarding improvements, the answers were unique – each participant effectively wished for different improvements, which ranged from a UI tweak (moving search closer to results) to different ways to compare submissions from different clusters. During the interview tasks, many also voiced out that they would like to use the system with a programming language their course used.

The participants in general agreed that the code search functionality was useful for finding interesting submissions, although its applicability was not clear for all – that is, simply searching code was not something that everyone found useful or meaningful. The majority considered that the modeling functionality could be useful (given potential improvements), while one participant was also neutral on this. In practice, this was a by-product of being uncertain whether there would be time to use the feature. All the participants agreed that sending collective feedback to students was useful.

Collective feedback based on the clusters was also found to have potential issues. Some participants noted that it was not clear that clustering could produce applicable and reliable clusters, even though this seemed like a more broad comment and not necessarily tied to CodeClusters. Similarly, the tuning of the model and setting parameters were seen as hard to grasp.

As a response to the question "What did you like the best about CodeClusters? Would you consider using it given improvements?", all participants expressed interest in trying the system out in the programming courses they were responsible for. Many recognized that the system had the potential of being a helpful tool for analyzing large quantities of submissions that otherwise would go unprocessed. It seems that simply clustering the submissions to gain an overview of most common patterns was by itself already valuable, while the possibility to provide collective feedback was in general a bonus. Every participant agreed that they could be providing additional feedback in programming courses with a system similar to CodeClusters.

## VI. Conclusion and Future Work

In this work, we explored the use of a system designed to cluster and review submissions for the purpose of providing feedback more efficiently. We presented CodeClusters and evaluated its use with five university lecturers from two universities. To summarize the evaluation, the results are mostly positive. Instructors expressed that learning to use the system within the duration of the interviews was challenging, which underlined a need for a good onboarding process. The features that CodeClusters provided were seen as quite satisfactory. Everyone expressed interest in using the system in their own course, given the availability of the programming languages that they teach and an integration with the LMSs that they use.

Our results indicate that the participants did not have similar existing systems that they use to analyze programming assignment submissions and that the designed system would provide opportunities for deepening knowledge about the students' programming patterns. We also observed some evidence in the participants perhaps not being too familiar with reviewing source code as none of them spotted a relatively simple mistake in a program. This could be due to the unfamiliar language, but it could be also due to a lack of practice with reviewing programming assignment submissions, which might be a by-product of being accustomed to using automatic assessment functionality.

Overall, CodeClusters offers a robust groundwork for further development, and while it became evident during the interviews that some UI enhancements are required to improve its usability, namely side-by-side comparisons between clusters, the basic approach appeared suitable for the task. The search was found to be a good general tool for finding outliers that would scale well to large datasets. The implemented similarity detection model, N-grams, while producing interesting clusters, could still require improvements to better discern higher-level patterns, especially in more complex assignments.

As a part of the future work, we are considering incorporating more features into the modeling process, such as CFGs, hierarchically generated N-grams, and additional metrics. These could result in improved accuracy in detecting meaningful structural patterns.

## Appendix

This appendix outlines the parts in the interview: a preliminary questionnaire, tasks given to the participants during the interview, and a satisfaction and system evaluation questionnaire.

### A. Preliminary questionnaire

The original questionnaire included options for some of the questions; these have been rephrased for conciseness. Questions 6

and 10-13 were Likert-like questions where the options ranged from 1 (Strongly agree) to 5 (Strongly disagree).

1) What is your current profession?
2) How many programming courses have taught or been part of teaching?
3) Over how many years have these courses been taught?
4) In these courses, how have the assignments mostly been graded?
5) In these course, how has the feedback on assignments been given to the students?
6) Grading assignments automatically is better than by hand.
7) If you had to grade an assignment by hand, what would be your process roughly?
8) What are the biggest problems related to giving feedback to students?
9) If you had to give feedback in writing to many students, do you think there would be a lot of repetitive work? Are there ways you would try to speed up the process?
10) Giving feedback to students has no positive impact on their performance.
11) It is very important to me to know how students have solved an assignment I have created.
12) I often check student submissions to see how they have approached a problem.
13) Giving good feedback can't be automated or it is too difficult to do so.

### *B. Interview tasks*

The interview tasks were presented in four parts. As a preliminary task, the participants were asked to open the address that hosted the system, to log in using specific credentials, and to choose a course and an assignment.

**Part 1: Searching**

1) Search code with the word "number". How many submissions do you find? How about using fuzzy search with edit distance of 2: "number 2"?
2) Clear the search field. Open CyclomaticComplexity facet. Scroll back to the main search bar and click the search button (rectangle with magnifying glass). Select the buckets with CyclomaticComplexity higher than 5. Run search again (you must do this always to update facet results). How many submissions do you find?
3) Reset the previous buckets and select CyclomaticComplexity bucket with value 3. Do you notice something strange with the submission?
4) Select all the submissions found in 1.3 and give them a review with a message: "1.4 message", metadata: "1.4 metadata" and tag "tag1".
5) Close all facets. Search submissions with the query "". Select the line with the  operator and add a review to the submission with the message: "1.5 You have used BITAND operator () instead of AND operator ().", "1.5 Fix tests to check for bitwise operators." and tags "tag1", "test-error".
6) Empty the search query. Add a custom_filter: BITAND_rare_keywords=[* TO *] Do you get the same result as previously? Delete the filter and add a new one: INT_keywords=1 How many submissions you find? Reset the custom filters.

**Part 2: Modeling**

1) Go to the Model tab, select N-gram model. Run the model without changing its parameters.
2) Select cluster with label 1. Quickly glance through the submissions. Then select cluster 2. Can you tell what is the main difference between the clusters?
3) If you select cluster 0 and quickly go through it, can you tell what is its difference to the others?
4) There should be a cluster with 16 submissions total. Select that cluster and all of its submissions and give them review "2.3 message" and "tag2".
5) Change the N-grams parameters minimum n-gram to 3 and maximum n-gram to 7 and the clustering algorithm to OPTICS. Run the model.
6) Select the cluster with 18 submissions. Select all submissions and add them to a review "2.5 message" and "tag2".

**Part 3: Review Flows**

1) Go to the Review Flows tab. Click New flow. Give the review flow title "3.0", description "3.0" and tag "tag3". Write * to the search query. Give review the message "3.0". Create the review flow.
2) Select the created review flow from the drop down. Click "Run and review". From the search results, find the submission with highlighted result "++oknumcount" and select the line. Give it a review with a message "3.1 Do you know what is the difference between ++variable and variable++?" and tag "tag3".

**Part 4: Accepting Reviews**

1) From the navigation bar, click the Reviews link to go to the /reviews page. Compare the submissions between review 2.3 and 2.5 and check the submissions that are missing from the other review. If you had to guess, what would be your explanation why.
2) Select those submissions and link them to the missing review using the "Link" button.
3) Accept all reviews. Filter the page to only show the SENT reviews.

### *C. Satisfaction and system evaluation questionnaire*

The original questionnaire included options for the second question. The question has been rephrased for conciseness. Questions 1, 4-6, and 9 were Likert-like questions where the options ranged from 1 (Strongly agree) to 5 (Strongly disagree).

1) CodeClusters was easy to use.
2) What were the biggest problems using CodeClusters?
3) If you could improve CodeClusters to fit your needs, how would you do it?
4) Having a code search was useful in finding interesting submissions.
5) Being able to run a model to cluster code wasn't very useful even considering future improvements.
6) Sending collective feedback to many students at once seemed useful.
7) I think the biggest problems in giving clustered feedback to many submissions at once are ...
8) What did you like the best about CodeClusters? Would you consider using it given improvements?
9) I could see giving feedback being integrated to programming courses using a system similar to CodeClusters.

## REFERENCES


[1] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Trans. Comput. Educ.*, vol. 19, no. 1, Sep. 2018.

[2] D. Boud and E. Molloy, *Feedback in Higher and Professional Education: Understanding it and Doing it Well*. Routledge, 2013.

[3] S. Gross, B. Mokbel, B. Hammer, and N. Pinkwart, "Towards providing feedback to students in absence of formalized domain models," in *Artificial Intelligence in Education*, H. C. Lane, K. Yacef, J. Mostow, and P. Pavlik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 644–648.

[4] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, "Overcode: Visualizing variation in student solutions to programming problems at scale," *ACM Trans. Comput.-Hum. Interact.*, vol. 22, no. 2, Mar. 2015.

[5] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: Scalable homework search for massive open online programming courses," in *Proceedings of the 23rd International Conference on World Wide Web*, 2014, p. 491–502.

[6] J. Huang, C. Piech, A. Nguyen, and L. Guibas, "Syntactic and functional variability of a million code submissions in a machine learning MOOC," *AIED 2013 Workshops Proceedings Volume*, vol. 1009, p. 25, 01 2013.

[7] H. Yin, J. Moghadam, and A. Fox, "Clustering student programming assignments to multiply instructor leverage," in *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, 2015.

[8] A. Vihavainen, M. Luukkainen, and M. Pärtel, "Test my code: An automatic assessment service for the extreme apprenticeship method," in *2nd International Workshop on Evidence-based Technology Enhanced Learning*, P. Vittorini, R. Gennari, I. Marenzi, T. D. Mascio, and F. D. l. Prieta, Eds. Heidelberg: Springer International Publishing, 2013.

[9] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *J. Educ. Resour. Comput.*, vol. 5, no. 3, p. 4–es, Sep. 2005.

[10] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, 2010, p. 86–93.

[11] R. Saikkonen, L. Malmi, and A. Korhonen, "Fully automatic assessment of programming exercises," *SIGCSE Bull.*, vol. 33, no. 3, Jun. 2001.

[12] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," *Computer science education*, vol. 15, no. 2, pp. 83–102, 2005.

[13] J. Carter, K. Ala-Mutka, U. Fuller, M. Dick, J. English, W. Fone, and J. Sheard, "How shall we assess this?" *SIGCSE Bull.*, vol. 35, no. 4, p. 107–123, Jun. 2003.

[14] R. Pettit and J. Prather, "Automated assessment tools: Too many cooks, not enough collaboration," *J. Comput. Sci. Coll.*, vol. 32, no. 4, p. 113–121, Apr. 2017.

[15] M. Brooks, S. Basu, C. Jacobs, and L. Vanderwende, "Divide and correct: Using clusters to grade short answers at scale," in *Learning @ Scale 2014*. ACM, March 2014.

[16] D. Liu and A. Petersen, "Static analyses in python programming courses," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, p. 666–671.

[17] P. Shah, M. Berges, and P. Hubwieser, "Qualitative content analysis of programming errors," in *Proceedings of the 5th International Conference on Information and Education Technology*, 2017, p. 161–166.

[18] J. Spacco, P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, and R. Duvall, "Analyzing student work patterns using programming exercise data," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 2015, p. 18–23.

[19] A. Vihavainen, T. Vikberg, M. Luukkainen, and J. Kurhila, "Massive increase in eager tas: Experiences from extreme apprenticeship-based CS1," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, 2013, pp. 123–128.

[20] S. Rogers, D. Garcia, J. F. Canny, S. Tang, and D. Kang, "Aces: Automatic evaluation of coding style," Master's thesis, EECS Department, University of California, Berkeley, May 2014.

[21] E. L. Glassman, R. Singh, and R. C. Miller, "Feature engineering for clustering student solutions," in *Proceedings of the First ACM Conference on Learning @ Scale Conference*, 2014, p. 171–172.

[22] A. Luxton-Reilly, P. Denny, D. Kirk, E. Tempero, and S.-Y. Yu, "On the differences between correct student solutions," in *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, 2013, p. 177–182.

[23] A. Damodaran, F. Di Troia, C. A. Visaggio, T. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *J. of Comp. Virology and Hacking Techniques*, 12 2015.

[24] I. Darwin, *Checking C Programs with Lint*, ser. C programming utility. O'Reilly & Associates, 1988.

[25] R. Roy Choudhury, H. Yin, J. Moghadam, A. Chen, and A. Fox, "Autostyle: Scale-driven hint generation for coding style," Master's thesis, EECS Department, University of California, Berkeley, May 2016.

[26] D. Kirk, T. Crow, A. Luxton-Reilly, and E. Tempero, "On assuring learning about code quality," in *Proceedings of the Twenty-Second Australasian Computing Education Conference*, 2020, p. 86–94.

[27] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel, "Scaffolding students' learning using test my code," in *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, 2013, p. 117–122.

[28] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann, "Writing reusable code feedback at scale with mixed-initiative program synthesis," in *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, 2017, p. 89–98.

[29] D. Weitekamp, E. Harpstead, and K. R. Koedinger, "An interaction design for machine teaching to develop ai tutors," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, p. 1–11.

[30] K. Rivers and K. R. Koedinger, "Automatic generation of programming feedback: A data-driven approach," in *The First Workshop on AI-supported Educ. for Comp. Sci. (AIEDCS 2013)*, vol. 50, 2013.

[31] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2007.

[32] K. Cooper and L. Torczon, *Engineering a Compiler*, ser. Morgan Kaufmann. Morgan Kaufmann, 2012.

[33] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: Association for Computing Machinery, 1970, p. 1–19.

[34] D. Hovemeyer, A. Hellas, A. Petersen, and J. Spacco, "Control-flow-only abstract syntax trees for analyzing students' programming progress," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 2016, p. 63–72.

[35] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *J. UCS*, vol. 8, no. 11, p. 1016, 2002.

[36] O. Karnalim and Simon, "Syntax trees and information retrieval to improve code similarity detection," in *Proceedings of the Twenty-Second Australasian Computing Education Conference*, 2020, p. 48–55.

[37] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, p. 308–320, Jul. 1976.

[38] J. Fitzpatrick, *Applying the ABC Metric to C, C++, and Java*. USA: Cambridge University Press, 2000, p. 245–264.

[39] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," 07 2012, pp. 111–121.

[40] C. Ragkhitwetsagul, "Code similarity and clone search in large-scale source code data," Ph.D. dissertation, University College London, UK, 2018.

[41] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, jun 2010.

[42] R. Ma and R. Angryk, "Distance and density clustering for time series data," in *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, 2017, pp. 25–32.

[43] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," *SIGMOD Rec.*, vol. 28, no. 2, p. 49–60, Jun. 1999.

[44] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1996.

[45] S. Basu, C. Jacobs, and L. Vanderwende, "Powergrading: a clustering approach to amplify human effort for short answer grading," *Trans. of the Association for Computational Linguistics*, vol. 1, pp. 391–402, 2013.

[46] A. Hevner and S. Chatterjee, *Design Research in Information Systems: Theory and Practice*, ser. Integrated Series in Information Systems. Springer US, 2010.

[47] L. T. M. Blessing and A. Chakrabarti, *DRM, a Design Research Methodology*, 1st ed. Springer Publishing Company, 2009.

[48] O. Seppälä, P. Ihantola, E. Isohanni, J. Sorva, and A. Vihavainen, "Do we know how difficult the rainfall problem is?" in *Proc. of the 15th Koli Calling Conference on Computing Education Research*, 2015, p. 87–96.

[49] T. Auvinen, V. Karavirta, and T. Ahoniemi, "Rubyric: An online assessment tool for effortless authoring of personalized feedback," *SIGCSE Bull.*, vol. 41, no. 3, p. 377, Jul. 2009.

[50] K. Heinonen, K. Hirvikoski, M. Luukkainen, and A. Vihavainen, "Using codebrowser to seek differences between novice programmers," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 2014, p. 229–234.