Making the Most of Repetitive Mistakes: An Investigation into Heuristics for Selecting and Applying Feedback to Programming Coursework

Roger Howell
Aston STEM Education Centre
Aston University
Birmingham, UK
howelrtc@aston.ac.uk

Shun Ha Sylvia Wong
Aston STEM Education Centre
Aston University
Birmingham, UK
s.h.s.wong@aston.ac.uk

***Abstract*—In the acquisition of software-development skills, feedback that pinpoints errors and explains means of improvement is important in achieving a good student learning experience. However, it is not feasible to manually provide timely, consistent, and helpful feedback for large or complex coursework tasks, and/or to large cohorts of students. While tools exist to provide feedback to student submissions, their automation is typically limited to reporting either test pass or failure or generating feedback to very simple programming tasks. Anecdotal experience indicates that clusters of students tend to make similar mistakes and/or successes within their coursework. Do feedback comments applied to students' work support this claim and, if so, to what extent is this the case? How might this be exploited to improve the assessment process and the quality of feedback given to students? To help answer these questions, we have examined feedback given to coursework submissions to a UK level 5, university-level, data structures and algorithms course to determine heuristics used to trigger particular feedback comments that are common between submissions and cohorts. This paper reports our results and discusses how the identified heuristics may be used to promote timeliness and consistency of feedback without jeopardising the quality.**

Keywords*—computer aided feedback, coursework assessment, static analysis, technology-enhanced learning

I. Introduction

Computer Science is a technical subject whose teaching ethos relies heavily on learning by doing [1]. The constructivist nature of learning programming [2] means that learning to program is an experiential and cyclical process. Such a learning process involves, broadly, experimentation to gain a concrete experience followed by several steps including an evaluation/reflection of that experience, and then the forming of plans or changing of views which then influence later actions/experiences [3][4]. To avoid continuous misconceptions and facilitate learning, it is important that students receive helpful and timely feedback on their work.

In the case of students completing programming coursework, feedback being returned from a tutor/assessor feeds into the reflection/evaluation stage of the learning cycle. As was noted by Hattie and Timperley [5, p.86], while the primary goal of feedback is "to reduce discrepancies between current understandings/performance and a desired goal", different strategies used may result in different levels of effectiveness. Wong and Beaumont [6] provided evidence that, for programming coursework, feedback must be timely, consistent, relevant, showing the locations of issues, and showing how to make improvements in order for it to be most helpful. However, due to the constraint of time, some tutors simply provide a mark and/or a brief overall comment as feedback. Therefore, it comes as no surprise that while student satisfaction levels on assessment and feedback in the UK has improved over the years, assessment and feedback remained a challenging area amongst UK Higher Education Institutions (HEIs) [7][8]. As shown by the latest National Student Survey (NSS) [7] with over 286,000 responses, 26% of respondents did not agree that marking had been fair; 27% of them did not agree that feedback on their work had been timely; and 25% of them did not agree that they had received helpful comments on their work. These figures are even higher when considering Computer Science student responses only.

To provide helpful and effective feedback is a particularly labour-intensive task for submissions of any non-trivial size. Timeliness and consistency in assessment and feedback generally deteriorate as the class size increases. Massively Open Online Courses (MOOCs) are an extreme example of large class sizes where there can be even tens of thousands of submissions to assess/provide feedback to. To address this problem across the range of tasks and class sizes, numerous computer-aided and computer-based tools have been produced to reduce this workload, e.g. Scheme Robo [9], BOSS2 [10], MarkTool [11], RubricAce [12] and EDM [13] to name a few.

As summarised by Keuning et al. [14], there are also a range of research into ways to provide feedback about a programming task to students. While computer-aided and computer-based marking tools and improved guidelines on providing feedback help, giving feedback to a large cohort of programming coursework that is helpful, timely, and consistent remains a challenging task.

The anecdotal experience from our colleagues indicates that clusters of students tend to make similar mistakes and/or exhibit similar good practices within their programming coursework. A computer-based marking tool, therefore, should not simply focus on supporting an assessor to evaluate each submission against a predefined set of marking criteria. It should provide more help for tutors to locate common issues amongst a cohort of submissions so that tutors can focus their effort on identifying misconceptions and constructing helpful feedback to each common issue so as to enable feedback given to students consistently and in a timely fashion.

We have examined feedback given to coursework submissions to a UK level 5, university-level, data structures and algorithms course to determine heuristics used to trigger particular feedback comments that are common between submissions and cohorts. This paper reports our results and discusses how identified heuristics may be used to promote timeliness and consistency of feedback without jeopardising the quality.

II. Related Work

This section explores techniques in which feedback grades and/or comments are applied to student coursework submissions. These techniques are split into four categories: comment reuse, automated tests, metrics, and source code similarity. A summary is included at the end of this section.

A. Comment Reuse

Feedback comment reuse is not a new idea. In its simplest form, comment reuse can involve the simple act of copying and pasting comments and phrases between files or perhaps the use of macros to insert pre-written comments. More sophisticated implementations of feedback comment banks include MarkTool [11] and eCAF [15], whose banks of comments can be dynamically added to and searched. In these cases, the feedback-giving is still predominantly manually-driven, relying upon human assessor(s) of a given set of submissions being aware of the comments within the comment bank and being able to locate them using matching terminology within search terms.

B. Automated Tests

The use of automated tests to provide immediate feedback is also not new, with Hollingsworth [16] making use of automated tests to validate student's punch card submissions as early as 1960. These simple tests involve verifying that, given a pre-determined set of inputs to a function/program, the expected results are returned. More recent implementations of software testing use a similar black-box input/output test approach and allow a significant boost in productivity of assessors with respect to the number of submissions that can be handled within a reasonable time-frame. In some cases, it is reported that thousands and even millions of unique submissions can be assessed fully independently [10][9][17] with the feedback being given to students including details about which test cases have passed or failed for a given set of inputs.

This dramatic improvement in the time required per-submission does not come for free, however, as it introduces the new requirement to adequately design tests for the relevant input/output combinations. In a relatively simple and repeated task such as a tutorial task, where the student submission may be only perhaps a dozen lines of code or perhaps the completion of a single function/class, the benefits of being able to scale up class sizes can far outweigh the costs of this up-front effort. The rise in MOOCs are an example of this for smaller specific tasks.

White-box analysis of the submitted code have also been utilised, evaluating how a given submission has completed a given task as opposed to just whether the correct output is produced. Yu et al. [18] gives the example of a task requiring the use of loops, where a student might hard-code the expected output using a series of print statements. In this case, automated black-box testing, such as unit testing, cannot adequately provide feedback on the techniques used by a student for that task.

Unfortunately, white-box testing techniques suffer similar concerns as black-box testing in that they require the assessor to adequately pre-empt the full range of possible (in)correct solutions beforehand in order to develop adequate test cases, and the feedback they return relates to the pass/fail of that particular test case. For example, Spacco et al. [17] notes that the pass/fail tests can be readily used to give feedback about low-level concepts such as syntactic correctness and the correctness of program output, though teaching assistants are still utilised to provide face-to-face feedback about higher level topics such as programming concepts and code style. Ultimately, full-automation of grading/feedback production using pre-defined test cases is only feasible for small and/or simple tasks whose solutions can be tightly and robustly defined.

C. Metrics

The Marker's Apprentice (TMA) [19], building upon CourseMarker [20], explores the use of rule-based code-correctness metrics to automatically determine the grade to be awarded to a submission. While some of these metrics are absolute (such as the presence of empty code blocks or incorrectly testing for equality), others require interpretation (such as determining the acceptable range of cyclomatic complexity or the total number of classes submitted). Pre-written feedback and guidance is then given about the rule being tested/violated, perhaps in the form of a link to the relevant documentation.

To enable automation, predefined ranges of acceptable values can be defined with the expectation that most correct solutions will utilise similar approaches therefore will have similar metrics, and incorrect solutions will be doing something unusual therefore will have atypical values. Determining the acceptable ranges for metrics can be somewhat problematic, particularly for large and/or complex tasks which may allow a broad range of acceptable submissions. While this use of metrics has been shown to be useful, it is not error-proof therefore using it in isolation is not advised. Instead, the use of metrics should be used as part of a suite of tools with a human assessor to provide some oversight and custom feedback comments, where possible [19].

D. Source Code Similarity

More recently, techniques relating to using source code similarity to enable the propagation of grades and feedback comments have emerged. AssignSim [21], for example, allows the human grader to mark a sample of submitted solutions and then have the system interpolate the grades of the un-marked submissions based on how similar they are to the manually-graded submissions. Similarity between submissions is determined within AssignSim by analysing the abstract syntax tree (AST) of submitted works and then comparing the AST against the ASTs of submitted works that have a known grade. While this is broadly effective and correlates well with grades awarded by human assessors, experimentation shows that the sample of manually-marked can skew the results for the full cohort and a priori selection (or pre-emptive creation) of a suitably representative sample is problematic. This technique does not appear to have been used to provide feedback comments.

OverCode [22] takes a slightly different approach in that it dynamically analyses program state during the execution of a set of pre-defined test cases. OverCode then aggregates and normalises submissions which have near-identical state, allowing the assessor to gain a broad overview of the techniques used and allows feedback comments to be written once and applied to all submissions within that aggregated cluster. While effective with small programming tasks (i.e. single functions perhaps a dozen lines in length) by a large cohort (>1000 submissions), it is unclear whether such an approach will be usable for larger programming tasks as the required program state analyses will be more complex and the range of potential solutions will be more diverse, making aggregation and normalisation of submissions with near-identical states a very challenging task.

Piech et al. [23] also considers the subject of feedback propagation, utilising neural networks to predict suitable feedback comments based on the AST sub-trees that are shared between submissions. This technique is then evaluated against a large set of over 200,000 submissions to a series of tasks which require only the completion of a simple function, with the use of an if/else function within a loop being described as the most difficult concept being evaluated. Again, while this is a technique that is promising for small and simple programming tasks, it is not yet suitable for medium-sized or coursework level tasks. Furthermore, as acknowledged by the authors, additional work is required to establish whether this approach will work effectively when there are limited submissions to guide the training of the neural networks. It is also unclear whether this approach will perform well on coursework submissions for more complex programming tasks with diverse solutions.

E. Summary

As shown within evaluations of the above tools, automated techniques such as software testing, static/dynamic source code analysis, and evaluating the similarity between submissions can be effectively used to scale the grading process. Where the range of acceptable/expected solutions can be effectively determined and this can be pre-emptively codified into a range of tests and/or metrics with an acceptable level of effort, these tools/techniques are particularly effective at grading.

TABLE I. THE COHORTS OF COURSEWORK SUBMISSIONS CONSIDERED IN THIS STUDY. COHORTS THREE AND FOUR WERE MARKED USING A NEW SYSTEM CURRENTLY UNDER DEVELOPMENT.

Cohort	**Cohort 1**	**Cohort 2**	**Cohort 3**	**Cohort 4**
Academic Year	2016/17	2015/16	2017/18	2016/17
Mode of Study	Full time, campus-based	Part time, distance learning	Full time, campus-based	Part time, distance learning
Class Size	118	30	127	37
Group Size	2-3	1	2-3	1
Number of Submissions Received	38	21	44	32
Marking System Used	eCAF	eCAF	*New System*	*New System*

Unfortunately, few tools can provide automated feedback to programming coursework tasks. For example, while eCAF [15] and other comment-banks can enable providing feedback to a task of any size, it is primarily human driven to write the comment and attach it to a submission. Tools such as TMA [19], the unnamed tool by Piech [23], and OverCode [22] can enable the application of comments to multiple submissions, their limitations include a significant up-front cost to pre-empt the range of possible solutions and to create tests/define metrics ranges specific to that task. Other tools focus upon the automated grading of programming task submissions.

III. Methodology

A. Data Source

To assess the potential for feedback comment reuse, we conducted a retrospective analysis of naturally-occurred feedback data. This feedback data was provided to four cohorts of students who submitted coursework to a Java-based data structures and algorithms module within a UK university. All students within this module have completed a substantial foundational object-oriented Java programming module as a prerequisite for studying this module. Table I gives an overview of the profile of each of the four cohorts whose coursework submissions were analysed within this study.

The coursework tasks set for each cohort were of a similar nature in that they require the student to create a standalone piece of object-oriented Java software that is able to parse and then model one or more, non-trivial, public-domain textual data. Their submitted solution was expected to model this data in a manner that enables efficient querying of this data. Note that each cohort were supplied with a different set of data files and functional requirements to minimise the risk of students sharing solutions.

The marking scheme for each cohort was also broadly similar in structure in that the marking criteria were grouped under five headings: (1) Overall Class Design, (2) Ability to Meet the Functional Requirements, (3) Ability to Meet the Non-Functional Requirements, (4) Program Design and

Algorithms Used, and (5) Program Presentation. Additional, more specific criteria were specified under these headings, forming a tree-structure to the marking scheme where the overall grade for each branch are, generally, the sum of the grades awarded to its children. While the marking criteria in sections 1, 4 & 5 were identical for each cohort, the marking criteria in section 2–3 differ amongst the cohorts as they were designed to suit the specific coursework task concerned.

Submissions comprised of typically 5-12 un-compiled Java source code files varying by the detail of the model and the use of any custom exception classes, each with typically 50-250 lines of code. In addition to this, submissions also included a UML class diagram describing their detailed class design in PDF or image form. These files were then submitted as a ZIP archive to the institution's Virtual Learning Environment (VLE), Blackboard [24], which forms the record of (i) the contents of the submission and (ii) the date/time it was submitted.

To grade and provide feedback to the students' work, submissions from cohorts 1&2 were marked using eCAF [15] while submissions from cohorts 3&4 were marked using a new electronic coursework assessment and feedback system developed internally. Both marking systems have similar functionality in that they both support the use of a hierarchical detailed marking scheme and feedback bank for assisting the marking process, and that both systems provide exportable feedback to students in the form of a downloadable HTML file. The feedback file details the grades awarded against each point of the marking rubric. It also contains a table containing: (i) the full text of the feedback comments, (ii) the precise location/range of the file(s) to which that comment was applied, and (iii) which element of the marking rubric the feedback comment relates to.

While the databases for each cohort were not retained between cohorts (except between cohorts 3&4), the exported HTML feedback files were stored/archived. The data used within this study was extracted from these HTML feedback files and imported into a Neo4j graph database [25], supported by an in-house tool developed for data analysis. This allowed queries to be used to extract/summarise the data, and then be imported into spreadsheet software for basic statistical and frequency analysis as described below.

B. *Analysis of Comment Reuse*

We first considered the reuse of feedback comments within a single cohort. This was calculated as the number of submissions which have received a uniquely phrased (distinct), comment at least once. Given that comments can be applied more than once to any given submission, only the first use of each comment was considered so as not to skew the data where an individual submission has repeatedly made the same mistake.

We then consider feedback comment reuse across cohorts. To enable this comparison, we first ranked the distinct comments by the number of submissions to have received that comment at least once and then normalised the number of comments by referring to "the top 10%" of comments and similar. The prevalence of a comment was similarly rated as a percentage of all comment uses, again only counting the first time it was used for a given submission. In this paper, we present the results as the proportion of distinct comments against the proportion of all comment uses.

To help identify feedback comments with essentially the same meaning, but differ by, say the variable name used in the submission, we then clustered/normalised the feedback comments given. As feedback comments can include several distinct sentences, the comments were first crudely split into comment fragments, roughly approximating sentences. Similarly-phrased fragments were then normalised into a consistent phrasing, and duplicate/ redundant fragments were deleted. Full comments were then clustered, being considered equal if they contained the same fragments in the same order. Where comments differed by only one fragment and the additional fragment was inconsequential to the meaning of the comment, full feedback comments were normalised to the shorter sequence of fragments.

C. *Analysis of Feedback Triggers*

For each feedback comment that appeared within multiple (five or more) different coursework submissions, we also examined the sections of Java code to which the comment had been applied to establish whether the underlying Java code fragments for each comment were also similar in nature. Where the linked fragments of code were similar, this provides evidence that the feedback comments were being given as a reaction to seeing something specific within the students' submitted work that the assessor wished to communicate to the learner. This enables us to identify the "trigger" for each feedback comment.

To examine the hypothesis that feedback comments are being reused due to elements of the submitted work being similar (e.g. either having the same praise-worthy element or showing the same error – described in this paper as a "trigger" for feedback), we cross-referenced the locations and ranges to which each comment was applied and analysed these code fragments for similarity. Specifically, we worked backwards from the most frequently reused comments to the least frequently reused. The code fragments to which these comments applied were then analysed and a set of key themes/topics that the comments relate to are then identified. The result of this thematic analysis is shown within Table III.

D. *Algorithmic/Programmatic Detection of Feedback Triggers*

Having identified a set of code samples which were identified as triggers for feedback comments, we then explored the ability to describe an algorithm to detect some of the identified trigger. The first stage involved describing the algorithm in terms that a human may be able to interpret and detect triggers, with a second stage beginning to explore automated detection.

IV. Results and Analyses

A. *Comment Reuse*

The data showed a relatively small proportion of distinct feedback comments accounting for a high proportion of the comments received by students. For instance, Table II shows that across the four cohorts of students, 475 uniquely-phrased (distinct) comments were used a total of 1189 times. This is a mean of 2.5 uses of each distinct comment. Limiting this to only comments which have been frequently re-used, in this case being used five or more times, Table II shows that 68 comments were used a total of 614 times. This is a mean of 9 uses per distinct comment.

Fig. 1 shows that the distributions of the proportion of distinct feedback comments were similar across all cohorts, with the top 20% of the most-frequently-reused distinct feedback comments accounting for between 40-60% of all comments used. During our analysis, it was noted that there appeared to be numerous comments which are phrased very similarly but are semantically identical.

One example of this included *"No initial capacity defined for this collection. This leads to repeated resizing."* versus *"Collections expected to contain many thousands of items have not been initialised with a specified initial capacity. This leads to unnecessary inefficiencies arising from collection resizing."*. Many more examples differed only by their punctuation, spelling, and the inclusion of detail specific to that particular task/submission such as referring to a particular variable or class name. In addition to variations based on the phrasing and punctuation of the comment, some comments also include non-specific phrases which are informational in nature rather than directly corrective or praising. One example of this was *"Please see the sample solution for an implemented example."* (and variations thereof).

TABLE II. A breakdown of the uniquely-phrased (distinct) comments given to students within each cohort, with comments counted only once per submission that it has been applied to. Total distinct count of comments is 452, as some comments are reused across cohorts.

	Appeared Within >= 1 Submission		Appeared Within >= 5 Submissions	
Cohort	***Distinct***	***All***	***Distinct***	***All***
Cohort 1	99	271	15 (15.2%)	135 (49.8%)
Cohort 2	51	92	10 (19.6%)[A]	44 (47.8%)
Cohort 3	172	412	17 (9.9%)	201 (48.8%)
Cohort 4	153	414	26 (17.0%)	234 (56.5%)
Total	*475*	*1189*	*68 (14.3%)*	*614 (51.6%)*

a. A – The top-10 frequently-used comments, as very few comments were used >= 5 times in Cohort 2.

TABLE III. Comment reuse across cohorts, before and after clustering of comments. Percentages are out of 454 distinct comments before clustering, and 331 distinct comments after clustering.

Comments reused across:	Before Clustering	After Clustering
4 cohorts	0% (0)	1.81% (6)
3 cohorts	0.22% (1)	1.51% (5)
2 cohorts	4.63% (21)	6.34% (21)
1 cohort	95.13% (430)	90.33% (299)
Total	***100% (454)***	***100% (331)***
Comments reused across at least two cohorts	***4.85% (22)***	***9.66% (32)***

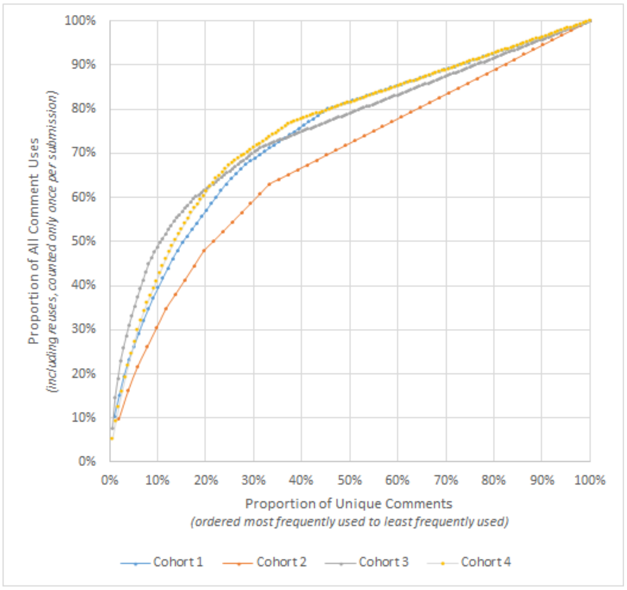

Fig. 1. Proportion of the total number of unique comments versus the proportion of the total uses of all comments (before clustering)

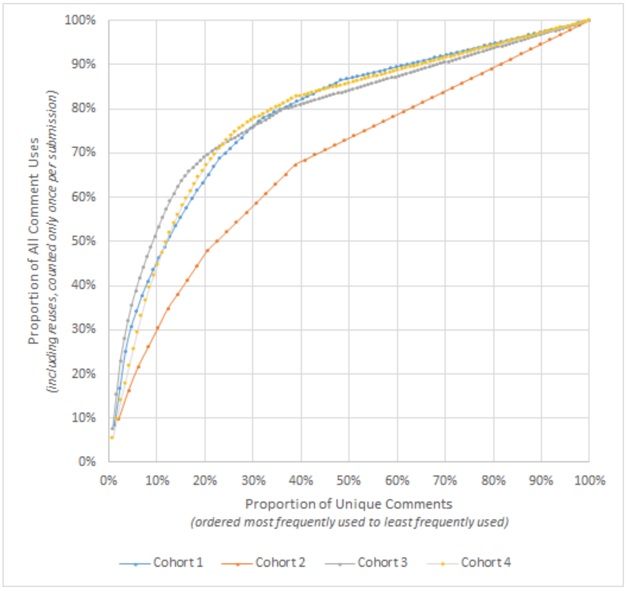

Fig. 2. Proportion of the total number of unique comments versus the proportion of the total uses of all comments (after clustering)

TABLE III. SUMMARY OF THE THEMES FOUND WITHIN EACH COHORT, GROUPED BY MARK SCHEME SECTIONS

Mark Scheme	Theme	Topic of comment	Cohort 1	Cohort 2	Cohort 3	Cohort 4
System Design, including UML class diagram	Diagram nomenclature	UML diagram mistakes	Y		Y	Y
		Incorrect realisation / implements arrow	Y		Y	Y
	Diagram content	UML diagram specificity (e.g. the inclusion of classes within the java.util / java.lang packages)			Y	
Ability to meet functional requirements	Specific functional requirement	Functioning with only a single file, rather than the required multiple input files	Y			
		Implementation (or not) of case-insensitive searches	Y			Y
		Not sorting output by popularity correctly		Y		
		Not implementing paths between stations			Y	
	Extensibility	Hard-coding of file references (outside of a main method)	Y			
		Inappropriate modification of the supplied skeleton code	Y			
	Robustness	(In)Correctly handling edge cases (e.g. null checks, file existence, invalid input data)		Y	Y	
Program design and algorithm(s) used, Including effective / efficient use of Java and OOP design (e.g. robust/extensible)	(In)Efficient use of Java classes / constructs	Good/Poor Initialisation of collections with an (in)appropriate initial capacity	Y	Y	Y	Y
		Use (or not) of a StringBuilder within a loop (as opposed to string concatenation)	Y		Y	
		Good/Poor Algorithm design (e.g. inappropriately reading files from disk on every interaction/query)	Y	Y		
	OOP / Program design	Good/Poor use of OOP design principles	Y			
		(In)Correctly overriding methods (e.g. compareTo/toString)		Y	Y	Y
		(In)Correct use of classes to model the input data (e.g. model does/doesn't correctly allow for multiple hypernyms)		Y	Y	Y
		(In)Efficient OOP design / indexing	Y	Y	Y	Y
	Program Output / Usability	(Un)Helpful error messages (e.g. just printing a stack trace to the console)	Y			
		(In)Complete program output		Y		
		Good/Poor use of error messages where no result found			Y	Y
		Good/Poor output formatting and clarity			Y	Y
	File handling	Good/Poor file processing technique (e.g. use of a BufferedReader for large input files)	Y		Y	
		Good/Poor text tokenisation (e.g. use of anchors within regular expressions)				Y
		Non-optimal and/or error-prone tokenisation (e.g. using multiple splits)				Y
Program presentation (of Java code)	Documentation comments	Good/Poor documentation comments		Y	Y	Y
		(In)Appropriate volume of comments (e.g. comments on virtually every line)			Y	
	Meaningful identifiers	Programming convention – (in)appropriate variable/class names (e.g. too short, misleading)				Y

In several cases, feedback comments differed only by the inclusion of this additional general fragment. Fig. 2 shows the result of normalising the set of comments applied to these cohorts. This figure shows that the first half of the curves are shifting up and to the left for most cohorts, with fewer comments being used only once or twice. The most frequently used distinct comments, therefore, now accounted for an increased proportion of all comment (re)use. Note that where the lines within Fig. 1 and Fig. 2 become linear, this represents the point where the comments used only once are being counted.

As shown within Table III, the reuse of feedback comments across cohorts was somewhat limited, with only 22 (of 454) distinct comments being used in multiple cohorts. This rose to 32 (of 331) distinct comments being used in multiple cohorts after normalisation/clustering of the comments. This is shown within Table III.

B. *Analysis of Feedback Triggers*

Our thematic analysis on the feedback comment data shows that, out of the 63 comments reused five times or more across any cohort's submissions, eleven themes emerged. Each theme is also composed of their own set of sub-themes/comment topics as shown within the aggregated result of this analysis in Table III. These comment topics relate to techniques and/or outputs that submissions either should or did utilise/contain.

Where a trigger is identified by a tutor within a submission and the exact location of trigger within the submission is highlighted, we now begin to examine if it is possible for a computer program to use these heuristics to search for other code fragments bearing the same characteristics within other submission files.

C. *Algorithmic/Programmatic Detection of Feedback Triggers*

Theoretically, all features of a submission referred to as comment topics can be described algorithmically. For example, comments under the topic "Not meeting a specific functional requirement", the triggers of those comments typically correspond to lacking certain expected programming routines, such as lacking an iteration routine to handle input from multiple files. The presence/absence of such programming routines may be programmatically detected via techniques including static code analysis such as searching for string patterns and dynamic runtime analyses such unit tests.

Practically, detecting the presence and/or absence of specific features in a submission can be very challenging. For example, with respect to UML class diagram themes/topics, such an algorithm may include the absence/presence of a specific style of arrow and/or a class. Programmatic detection of such features in the submissions is not straight-forward, due to the range of submission formats (image/pdf) and the complexity involved in programmatically analysing such files.

To investigate the effectiveness of programmatically detecting specific coding features in text files and the challenges behind this task, we have carried out a preliminary study into using regular expressions to help identify code features around the theme of "(In)Efficient use of Java classes / constructs" on Cohorts 3 and 4. One typical issue noted in the feedback comments analysed in this study was that the submissions did not include appropriate initialisation of array-based collections. To identify submissions with this issue programmatically, we used a regular-expression-based pattern matching approach to identify the locations in each submission where such issue occurred. Note that the absence of an initial capacity specification may be appropriate in some situations. The results were then manually verified to establish their correctness in the given context. Manual analysis of the matches to this regular expression are included below, where we show a breakdown.

Our regular expression experiment identified a total of 599 locations in the two cohorts of submissions where an initial capacity specification may appear in the submitted Java code, as described in Table IV. Only two instances (0.3%) are where the defined regular expression matches were. Of the remaining 597, 155 matches had a human-applied comment attached to that fragment relating to the initialisation of collections with an appropriate initial capacity (a mixture of praise and guidance). Of the 444 matches which did not have a human-applied comment applied, approximately a quarter (103) relate to a case where a positive or critical comment may have been warranted due to it clearly being an appropriate or inappropriate initial capacity, with the remaining 339 being debateable with no feedback comment being applicable (e.g. benefit of the doubt may be given that the default capacity of 16 items is the desired initial capacity). Our results show that 17.3% of the identified locations in the submissions could have been given a feedback comment, but were missed from the manual marking process.

TABLE IV. A BREAKDOWN OF THE AUTOMATED PATTERN MATCHES IN COHORT 3 AND COHORT 4, COMPARING WHETHER A RELEVANT COMMENT HAS BEEN APPLIED TO THE FRAGMENT OF CODE IDENTIFIED BY THE AUTOMATED REGULAR EXPRESSION MATCH. NOTE: ONE "NOT APPLICABLE" MATCH RELATES TO A SPURIOUS NAMING MATCH, AND THE SECOND "NOT APPLICABLE" MATCH RELATES TO A FRAGMENT OF CODE THAT HAD BEEN COMMENTED OUT BY THE SUBMITTER PRIOR TO SUBMISSION.

(Cohorts 3 and 4) Collection Initialisation Code Fragment with:	**Initial Capacity Comment Manually Applied by the Assessor?**		
	No Human Comment	***Human Comment***	***Total***
Inappropriate Initial Capacity	74 (56%)	57 (44%)	131 (100%)
Debateable Initial Capacity	339 (86%)	55 (14%)	394 (100%)
Appropriate Initial Capacity	29 (40%)	43 (60%)	72 (100%)
Not Applicable	2 (100%)	0 (0%)	2 (100%)
Total	*444 (74%)*	*155 (26%)*	*599 (100%)*

V. DISCUSSION

While feedback banks can help promote consistency and reduce the time required in drafting feedback comments, the onus remains with assessors to identify where within a coursework submission a feedback comment should be given. With a non-trivial coursework task and a large cohort of submissions, this is a difficult task for tutors to achieve in a thorough and complete manner.

Our feedback data analysis shows extensive feedback comment reuse, therefore repeated work. As a specific example, over 50% of all feedback comments given to each cohort of submissions come from only approximately 20% of the unique comments given to the submissions (cf. Fig. 1 & Fig. 2). If a means to (semi-)automatically identify different instances of the same trigger within the remaining submissions of a cohort were to exist, once a tutor had identified the cause of needing to give that particular comment then the consistency and timeliness of feedback can be further promoted. If we were to measure the work required to assess a programming coursework as the number of feedback comments assigned to each coursework submission, then we believe that such automation would result in significantly less work for similar output.

Using feedback data given across four cohorts of students using the Java programming language to complete a Data Structures and Algorithms coursework task, we have identified 11 themes of feedback triggers. Many of these feedback triggers would allow a tutor to give more consistent feedback to students without the need to spend a long time in manually identifying all praise-worthy features and/or concerning issues within each coursework submission. In this respect, the focus here would be in empowering the assessor rather than replacing the assessor via fully automatic assessment techniques. By freeing the time that a tutor would normally spend authoring/searching for feedback comments, the tutor can then spend more time on crafting helpful feedback to their students.

VI. Conclusion

As this is a retrospective analysis of naturally-occurring data and that no specific feedback bank had been set up to aid assessment of submissions across all 4 cohorts, overlaps in comment reuse between cohorts naturally relatively low. It is noted that if the same feedback bank were to have been shared, the frequency of feedback reuse would have been even higher.

Our preliminary results using simple regular expressions show that this approach has potential, though additional work is needed to refine the algorithmic definitions of feedback triggers and to establish the effectiveness of this approach across a wider range of comment triggers.

In this paper, we have presented our work in analysing the extents of feedback reuse in a typical data structure and algorithms programming coursework. Our results show that similar issues and praise-worthy elements can be found amongst a cohort of coursework submissions which leads to a fair amount of reuse in feedback comments.

Our thematic analysis on the assessment feedback given to four cohorts of coursework submissions indicates that it is possible to look for a known trigger of feedback comment amongst all submissions once the first instance of that trigger has been identified in a submission.

Preliminary experimental results show that defining triggers of feedback comments using regular expressions are promising as a means of programmatically detecting some triggers for feedback. However, further work is required to evaluate its effectiveness on a wider range of code features and to make it an accessible approach to those who are not au-fait with regular expressions. Additionally, as a non-Turing-complete language, regular expressions are limited in what they can detect – particularly with nested bracket patterns – where alternative approaches such as using syntax tree queries and patterns may be more appropriate.

References

[1] O-H. Ylijoki, “Disciplinary cultures and the moral order of studying – a case-study of four Finnish university departments,” *Higher Educ.*, vol. 39, no. 3, pp. 339–362, 2000.

[2] M. Ben-Ari. “Constructivism in Computer Science Education,” *J. of Comput. in Math. and Sci. Teaching*, Norfolk, VA, vol. 20, no. 1, pp. 45–73, 2001.

[3] D. A. Kolb, *Experiential Learning: experience as the source of learning and development,* London:Prentice-Hall, 1984.

[4] D. A Kolb, *Experiential learning: Experience as the source of learning and development*, FT press, 2014.

[5] J. Hattie and H. Timperley, “The Power of Feedback,” *Review of Educational Research*, vol. 77, no. 1, pp. 81-112, 2007. doi: 10.3102/003465430298487

[6] S. H. S. Wong, and A. J. Beaumont, “A Quest for Helpful Feedback to Programming Coursework,” *Engineering Educ.,* vol. 7, no. 2, pp. 51-62. doi: 10.11120/ened.2012.07020051

[7] Higher Education Funding Council for England (HEFCE) (2017), *2017 National Student Survey (NSS) results by teaching institution for all institutions*, [Online]. Available: http://www.hefce.ac.uk/lt/nss/results/2017/

[8] P. Surridge, “The National Student Survey 2005-2007: Findings and Trends – A Report to the Higher Education Funding Council for England,” University of Bristol, 2008

[9] R. Saikkonen, L. Malmi, A. Korhonen, “Fully automatic assessment of programming exercises,” in *Proc. 6th annual SIGCSE Conf. Innovation and Technology in Comput. Sci. Educ. (ITiCSE),* Canterbury, United Kingdom, *vol.* 33, 2001, pp. 133-136. doi: 10.1145/377435.377666

[10] M. Joy, N. Grifths, and R. Boyatt, “The BOSS online submission and assessment system,” *J. on Educational Resources in Computing (JERIC),* vol. 5, no. 3, 2005. doi: 10.1145/1163405.1163407

[11] E. Heinrich, J. Zhang, “A system designed to support formative assessment of open-ended written assignments,” in *5th IEEE Int. Conf. Advanced Learning Technologies (ICALT),* Kaohsiung, Taiwan, 2005, pp. 88-92. doi: 10.1109/ICALT.2005.29

[12] N. Wiratunga, I. Adeyanju, P. Coghill, and C. Pera, “RubricAce (TM): A case-based feedback recommender for coursework assessment,” in *Proc. of the 16th UK Workshop on Case-Based Reasoning,* vol. 829, 2011.

[13] E. Albrecht and J. Grabowski, “Towards a Framework for Mining Students’ Programming Assignments,” in *IEEE Global Engineering Educ. Conf.*, Norfolk, VA, 2016, pp. 1096–1100. doi: 10.1109/EDUCON.2016.7474690

[14] H. Keuning, J. Jeuring, and B. Heeren, “Towards a Systematic Review of Automated Feedback Generation for Programming Exercises – Extended Version. Technical Report,” Department of Information and Computing Sciences, Utrecht University, UU-CS-2016-001, 2016.

[15] S. H. S. Wong, J. Taylor, and A. J. Beaumont, “Enhancing Student Learning Experience through a novel Electronic Coursework Assessment and Feedback Management System,” in *Proc. for EE2012 - Innovation, Practice and Research in Eng. Educ.*, Coventry, UK, 2012, pp. 18–20.

[16] J. Hollingsworth. “Automatic graders for programming classes,” *Commun. ACM (CACM),* vol. 3, no. 10, 1960, pp. 528–529. doi: 10.1145/367415.367422

[17] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez, “Experiences with marmoset,” in *Proc. 11th annual SICGSE Conf. on Innovation and Technology in Comput. Sci. Educ. (ITiCSE),* Bologna, Italy, 2006, pp. 13-17. doi: 10.1145/1140124.1140131

[18] Y. T. Yu, C. M. Tang, C. K. Poon, and J. W. Keung, “Adoption of Computer Programming Exercises for Automatic Assessment – Issues and Caution,” in *25th Int. Conf. on Comput. in Educ.*, 2017, pp. 555-564.

[19] S. Nutbrown and C. Higgins, “Static analysis of programming exercises: Fairness, usefulness and a method for application,” *Comput. Sci. Educ.*, vol. 26, no. 2-3, pp. 104-128, 2016. doi: 10.1080/08993408.2016.1179865

[20] C. Higgins, T. Hegazy, P. Symeonidis, and A. Tsintsifas, “The CourseMarker CBA System: Improvements over Ceilidh,” *Educ. and Information Technologies*, vol. 8, no. 3, pp. 287–304, 2013. doi: 10.1023/A:1026364126982

[21] K. A. Naudé, J. H. Greyling, and D. Vogts, “Marking student programs using graph similarity,” *Comput. & Educ.*, vol. 54, no. 2, pp. 545–561, 2010. doi: 10.1016/j.compedu.2009.09.005

[22] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. “OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale,” *ACM Trans. Comput.-Human. Interaction – Special Issue on Online Learning at Scale (TOCHI),* vol. 22, no. 2, pp. 7:1-7:35, 2015. doi: 10.1145/2699751

[23] C Piech *et. al*, “Learning program embeddings to propagate feedback on student code,” in *Proc. 32nd Int. Conf. Mach. Learning*, Lille, France, vol. 37, 2015, pp. 1093-1102.

[24] Blackboard Inc. (2018), *Blackboard VLE,* [Online]. Available: http://www.blackboard.com/index.html

[25] Neo4j, Inc. 2018. *The Neo4j Graph Platform.* [Online]. Available: https://neo4j.com/