Examining Student Coding Behaviours in Creative Computing Lessons using Abstract Syntax Trees and Vocabulary Analysis

Matthew Yee-King
Goldsmiths, University of London
London, UK

Louis McCallum
Goldsmiths, University of London
London, UK

Maria Teresa Llano
Goldsmiths, University of London
London, UK

Vit Ruzicka
Creative Computing Institute,
University of the Arts
London, UK

Mark d'Inverno
Goldsmiths, University of London
London, UK

Mick Grierson
Creative Computing Institute,
University of the Arts
London, UK

ABSTRACT

Creative computing is an approach to computing education which emphasises the creation of interactive audiovisual software and an art-school influenced pedagogy. Given this emphasis on Dewey's "learning by doing", we set out to investigate the processes students use to develop their programs. We refer to these processes as the students' 'coding behaviour', and we expect that understanding it will provide us with valuable information about how students learn in our creative computing classes. As existing metrics were not sufficient, we introduce a new set of quantitative metrics to describe coding behaviours. The metrics consider factors such as students' vocabulary use and development, how fast and how much they alter the functionality of code over time and how they iterate on their code through text insert and delete operations. Many of our lessons involve providing students with demonstrator code which they use as a base for the development of their programs, so we use demo code as an entry point to our dataset. We look at programs students have written through developing the demo code in a dataset of over 16,000 programs. We clustered the demo code using the set of descriptive metrics. This lead to a set of clusters containing programs which are associated with distinct coding behaviours. Four was the ideal number of clusters for cluster density and separation. We found that the clusters had distinct behaviour patterns, that they were associated with different instructors and that they contained demo programs with different lengths.

CCS CONCEPTS

• **Human-centered computing → Human computer interaction (HCI)**; *Usability testing*; Heuristic evaluations.

KEYWORDS

Demonstrator Code; Creative Computing; MOOCs; Automated Code Analysis

ACM Reference Format:
Matthew Yee-King, Louis McCallum, Maria Teresa Llano, Vit Ruzicka, Mark d'Inverno, and Mick Grierson. 2020. Examining Student Coding Behaviours in Creative Computing Lessons using Abstract Syntax Trees and Vocabulary Analysis. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20), June 15–19, 2020, Trondheim, Norway.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3341525.3387408

1 INTRODUCTION

Creative computing is a branch of computing that emphasises the development of software that generates interactive, audio and graphical output. It embraces an arts-inflected pedagogical approach which encourages students to engage in Dewey's 'learning by doing'[7].

An important element of creative computing pedagogy is the idea that students experiment with code, driven by technical and aesthetic goals. The *process* of creating programs is therefore critical, and this inspired us to create a set of metrics that allow us to describe and examine that process.

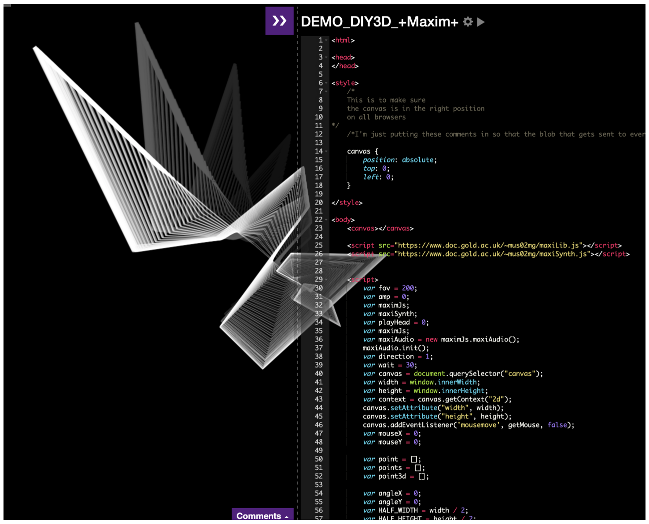
When we are teaching creative computing we often use demonstrator code. Demonstrator, or demo code is code that illustrates a technical or aesthetic concept, for example, an audio demo program showing how to load and play a drum sound, or an audiovisual program showing how to visualise the audio signal detected by a microphone.

Many of our creative computing lessons involve showing students demo code, then instructing them to take that code and adapt it in different ways, but we currently have no way of determining the types of student learning behaviours occurring following these lessons. Does our demo code allow students to set off and explore? Are they able to develop it into new programs, different from the original? Investigating this is made harder if the teaching is done via Massive Open Online Courses (MOOCs) with a large, remote community of students and creators.

We investigate this problem by looking in detail at what students do with our demo code after we give it to them. We can do this because we have created a browser-based, creative coding platform which thousands of our students have used to develop code, and which contains a dataset of about 25,000 programs, from which we analyse 16,245 in this paper. We developed the platform to support our teaching needs. The platform allows us to rapidly share code

ITiCSE '20, June 15–19, 2020, Trondheim, Norway

ACM ISBN 978-1-4503-6874-2/20/06…$15.00
https://doi.org/10.1145/3341525.3387408



```
<html>

<head>
</head>

<style>
    /*
    This is to make sure
    the canvas is in the right position
    on all browsers
*/
    /*I'm just putting these comments in so that the blob that gets sent to ever

    canvas {
        position: absolute;
        top: 0;
        left: 0;
    }

</style>

<body>
    <canvas></canvas>

    <script src="https://www.doc.gold.ac.uk/~mus02mg/maxiLib.js"></script>
    <script src="https://www.doc.gold.ac.uk/~mus02mg/maxiSynth.js"></script>


    <script>
        var fov = 200;
        var amp = 0;
        var maximJs;
        var maxiSynth;
        var playHead = 0;
        var maximJs;
        var maxiAudio = new maximJs.maxiAudio();
        maxiAudio.init();
        var direction = 1;
        var wait = 30;
        var canvas = document.querySelector("canvas");
        var width = window.innerWidth;
        var height = window.innerHeight;
        var context = canvas.getContext("2d");
        canvas.setAttribute("width", width);
        canvas.setAttribute("height", height);
        canvas.addEventListener('mousemove', getMouse, false);
        var mouseX = 0;
        var mouseY = 0;

        var point = [];
        var points = [];
        var point3d = [];

        var angleX = 0;
        var angleY = 0;
        var HALF_WIDTH = width / 2;
        var HALF_HEIGHT = height / 2;
```

Figure 1: The browser-based coding platform. On the left is the visual output of the program. On the right is the code editor.

with students who can then make copies or forks of the code. Later we can reconstruct coding sessions to analyse them.

Creative computing has its own set of teaching approaches and learning outcomes distinct from other forms of computer science education so finding metrics that can identify different student coding behaviours in this context provides its own set of challenges. If we can overcome these challenges and develop a set of descriptive metrics, we can use them to improve our teaching. For example, we can analyse how students work with our lesson materials to see if they are able to explore and expand them in the way we expect, or we might develop automatic marking systems that can identify students who carry out an effective creative computing process.

With this in mind, to analyse the patterns of use on the coding platform, we have developed a set of metrics based on automatic analysis of how demonstrator code is iteratively forked and edited by students. We have selected the metrics to describe some of the kind of learning activity often displayed in creative computing contexts. The metrics measure aspects of coding behaviour such as the speed and amount at which the program functionally changes, how the coder uses existing and new vocabulary and how they experiment via insert and delete operations.

In the analysis presented here, we take demo code as an entry point to the dataset. We can then associate the coding behaviours we identify with particular examples of demo code. The demo code from which students developed their programs is only one aspect of the context in which the coding took place though. Other contextual elements include the instructor giving the lesson, the style of teaching and the learning objectives.

We find that our metrics can be used to identify different clusters of coding behaviours, each associated with different characteristics relating to how the demonstrator code is taken and edited by students. Interestingly, these coding behaviours seem to be associated with specific instructors, some of whom authored the code with distinct pedagogical intentions for a particular context.

In the following text, we present our approach to identifying student coding behaviours from a large dataset of creative computing programs derived by students from demonstrator code.

2 RELATED WORK

2.1 Teaching with Demonstrator Code in Computer Science Education

Demonstrator and example code provides a resource for learning about new programming concepts and libraries [19] and is often the starting point when developers start new projects [4]. Brandt et. al describe the approach of opportunistic programming, where already existing code snippets are scavenged and reused by developers, often via copy-and-paste actions [5]. Demonstrator code may be authored and presented with a variety of different intentions for how students will develop it to engage with the learning outcomes of a particular class. One strategy that is often encouraged in creative computing is a STEAM approach, where students take an active, exploratory approach to lessons [23]. This is also seen more widely in the general teaching of programming where, following on from Dewey's 'learning by doing' [7], much research has been conducted into the benefits of this approach to learning involving trial and error, hacking and building things without necessarily knowing exactly how they work [1, 8, 15, 17, 20].

For example, in a task developing virtual soccer playing robots, Berland et. al find that students often undertook an "explore, tinker, refine" approach, successfully using trial and error in an environment designed to allow quick evaluation and low risk failure to arrive at solutions [2]. Yee-King et al. have also found that encouraging an exploratory approach when teaching creative computing has positive effects on student experience and that students partaking in such tasks edit their programs in particular, identifiable ways [23]. Further, in a study of Scratch projects, Dasgupta et. al demonstrate that the more users remix projects, that is, take an existing project as a base for building a new project, the larger their programming vocabulary [6], and that when computational concepts are presented in the context of remixing, users are more likely to use them in future.

2.2 Automated Analysis of Student Code

Code analysis is a broad research area with many different applications, for instance, protecting intellectual property, predicting code errors, debugging function calls, identifying undocumented code and identifying semantic similarities between two pieces of code. In this work we are interested in automated analysis of students' code. Two prevalent themes of research in the area are clustering similar submissions together and modelling a student's code trajectory over time. Whilst neither of these aligns directly with our specific aim, both provide useful metrics by which we might characterise a demo, or by which we might track its evolution over time.

MOOCs have motivated some recent work analysing student code. Due to the volume of submissions and the low cost of the courses, it can be beneficial to cluster similar coding assignments together and to provide group feedback. Yin et al. [24], for instance, use a measure of Tree Edit Distance (TED), calculated from Abstract

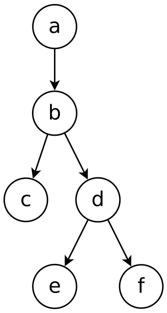


Figure 2: Relationship between items in the fork tree where circles represent programs on the platform. 1) a is the root of all nodes 2) b forks a, d forks b. 3) b,c,d,e and f are descendants of a. 4) d,b and a are ancestors of f. 5) b, not a or d is the most important ancestor of f as it is the oldest and has the most forks.

Syntax Trees (ASTs), and t-distributed stochastic neighbor embedding (t-SNE) to cluster and visualise similar pieces of code. Huang et al. [12] specify different types of similarities, namely functional and syntactic. Functional similarity comes through comparing output vectors of unit tests, while syntactic similarity is measured by comparing tree edit distance for ASTs.

More sophisticated approaches, such as that of Glassman et al's OverCode system [11], not only cluster similar submissions, but are also able to generate a summary code representative of each cluster, allowing for instructors to view more submissions and get a better feel for the breadth and variety of their content. Having a general understanding of the spectrum of solutions provided by students not only facilitates instructors' marking but also enables them to provide better feedback.

A further motivation for automatic code analysis is to track students' progress over time. Piech et al. [18] log the code at each compilation and analyse the student's trajectory by constructing Hidden Markov Models based on bag-of-words, difference in API calls and AST change severity features. They then use dynamic time warping to align records, identifying students with similar trajectories by clustering the HMMs. Following on from this, Blikstein et al. [3] found that it was the change in programming patterns over time, rather than any particular programming pattern, that was most predictive of the final grade. Studying informal learning in Scratch, Yang et al. studied the learning trajectories of students across multiple projects, using clustering to find 4 separate approaches [22]. Xie and Ableson tracked the progression of individual learners using a different online block-based programming environment and reported the way users added new blocks to their vocabulary changed as they made more projects [21].

3 METHOD

In this section we will describe the dataset and the coding behaviour metrics we have extracted from it. We will then explain how we went about clustering and examining the coding behaviours.

3.1 The Dataset

We gathered the dataset analysed in this study from our browser-based coding platform, the interface of which is shown in Fig 1. We had initially developed the platform to meet our teaching requirements for a creative computing BSc curriculum as existing platforms such as codepen.io[1] and JSFiddle[2] could not meet these requirements at the time. The key features of the platform include JavaScript-based development, with built-in support for audiovisual signal processing libraries, document sharing, code forking, collaborative code editing and live-coding.

The platform is underpinned by an operational transform engine[9]. As a consequence, every edit (transformation) of the program by the user is recorded, in order. Collecting data this way allows us to reconstruct the programming process step by step.

Several different cohorts of students have used the platform, including MOOC learners, on-campus university students and those attending in-person short course workshops with all the courses broadly based around themes of audiovisual processing and creative computing. We also used the platform as a way to share new libraries developed in research projects with communities of public users. In total, the platform contains around 25,000 programs.

3.1.1 Selecting programs for analysis. In our lessons, we often share demonstrator code with students, and they make copies of this code which they continue to develop. We call the copies forks, and we call the tree of forked programs following on from a particular demo code item the fork tree (see Figure 2).

In the analysis presented here, we selected every program on the platform with at least five descendants made by other users in its fork tree. This process yielded 304 root programs, which we refer to as demo programs. Then we compute the metrics described below over the 304 demo programs' fork trees, which involved analysing a further 16,245 programs.

The rationale was that we considered these items to be clear starting points for code development by others, and thus a key element of the context in which that coding took place. Thus we can examine if that context, i.e. the demo code itself has characteristics which enable (or inhibit) certain coding behaviours.

While forking was encouraged by instructors and is a crucial part of the platform, we accept that our fork tree analysis method does not necessarily capture certain learning approaches. For example, students might copy-paste code from other documents into a fresh, unforked program as a method of exploring the lesson content, as opposed to forking. Our method does represent copy-paste operations within forked code though. We acknowledge that the method of analysing coding behaviours does not explicitly take account of other aspects of the lesson context besides the demo code, for example, which instructor taught the lesson, the learning objectives, if it was online or face to face teaching, and so on. However, in our analysis, we do show how we can map back from coding behaviours to other contextual elements by correlating instructors with clusters. The mapping from clusters to instructors is described more in the Section 4.

[1] https://codepen.io
[2] https://jsfiddle.net

3.2 Coding behaviour metrics

We shall now describe the metrics we extracted from the code editing sequences in our dataset.

3.2.1 Forks by other users. This metric is the average number of descendants per user created by users other than the original program creator. It tells us how much other people re-use a program. The averaging per user allows comparison between programs shared with different sized cohorts. Higher forks per user indicate potentially greater re-use.

3.2.2 Code editing rate. We calculate the rate of code editing in each document as operations per hour, ignoring periods of inactivity longer than five minutes. An operation is an insertion or deletion in the code, including copy/cut and paste operations within the document. We then traverse the fork tree and compute the average rate over all the documents, only considering documents created by other people. Higher scores indicate higher code editing rates.

3.2.3 Code editing amount. We calculate the total number of edit operations per document in the fork tree, then take the average per document, only considering documents created by others. Higher scores indicate higher numbers of edits.

3.2.4 Insert to delete ratio. We calculate the ratio between insert operations (adding code to a document) and delete operations (removing code from a document), then take the average over documents in the fork tree. We use a 1:1 ratio as a proxy for exploratory coding, or 'tinkering' as described by Papert[15].

3.2.5 Syntax tree change rate. This metric makes use of an AST representation of the program. AST diffing allows a more meaningful comparison of two programs at a functional level than regular text diffing does. AST diffing can detect moved and renamed elements as well as the more obvious inserted and deleted elements.

For each document, we reconstruct a chronological sequence of executable snapshots of the program code. We compare each snapshot to its previous snapshot in the AST domain and use the Gumtree diffing algorithm to compute a difference score [10]. From this, we can calculate the diff rate, ignoring periods of inactivity longer than five minutes. We compute the mean diff rate per hour over the fork tree.

We adjust the AST diff rate by dividing the code editing rate by it to prevent large copy-paste operations which induce very rapid AST change from out-ranking smaller, handmade code edits. This biases the metric to rate small, exploratory or iterative edits over copy-past edits.

3.2.6 Syntax tree change amount. Syntax tree change amount is the total amount of AST change observed in the lifetime of a document, from when the user forked it, to its last edit. We average this over documents in the fork tree.

3.2.7 Vocabulary re-use. We define the vocabulary used in a program as the set of unique function calls. Vocabulary re-use is when the function call appears more times in the descendant than in the ancestor - the user used that function again. We compare descendants to their most important ancestor instead of the root. The most important ancestor is the ancestor with the most forks, then the oldest in case there are more than one with the same number of forks, as illustrated in figure 2. We use this instead of the root ancestor because often in our teaching we will fork all of our examples from a starter program, then share the forks with students, and we are most interested in comparing student code with the actual examples given out in class, not just the root of all examples. We average this metric over all documents in the fork tree to avoid rewarding documents shared with large classes over documents shared with small classes.

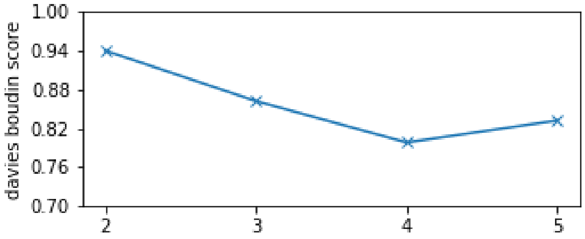


Figure 3: Davies-Baudin score for different values of k

3.2.8 New vocabulary. New vocabulary is the set of function calls that appear in the descendant but not in the ancestor, where the user added new function calls to the program. We average over the documents in the fork tree.

3.2.9 Fine-tuning. Having first defined the metrics, we then began iterating back to fine-tune them as we observed quirks in the data. For example, we noticed a document with high AST change rate also had very few edit operations, and concluded it involved lots of copy-paste instead of fine-grained editing. To correct this we implemented a normalised AST diff rate. This is not to say that large copy-paste operations are not necessarily a useful indicator of coding behaviours, merely that this metric was intended to capture fine-grained editing as opposed to large copy-paste operations.

3.3 Clustering demo code

To identify specific coding behaviours, we used the above metrics to cluster the 304 demonstrator programs (programs with 5 or more descendants) by extracting the metrics from the 16,245 descendants of that demo code and calculating the mean metrics per demo program.

After reducing the dimensionality of the features from 8D to 2D using a Principle Component Analysis (PCA), we used the K-means++ algorithm from the scikit-learn library [16] to cluster the programs with k = 4.

This value of k was chosen because of its low Davies-Baudin score, representing a better separation between the clusters [14], after which it starts to rise again (see Fig 3). We then take the clusters and look at the mean values for each metric, the percentage of programs written by particular known, prolific instructors and the overall length to try and characterise the types of programs present in each and their associated coding behaviours.

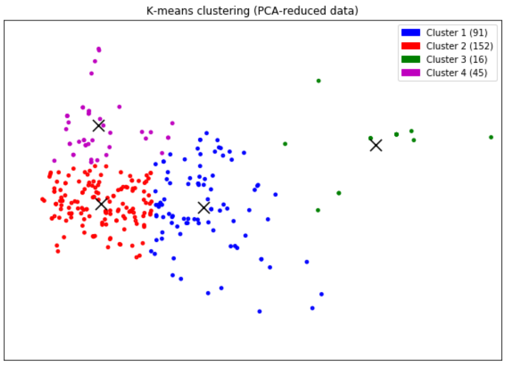


Figure 4: Clusters of programs, centers denoted by black X

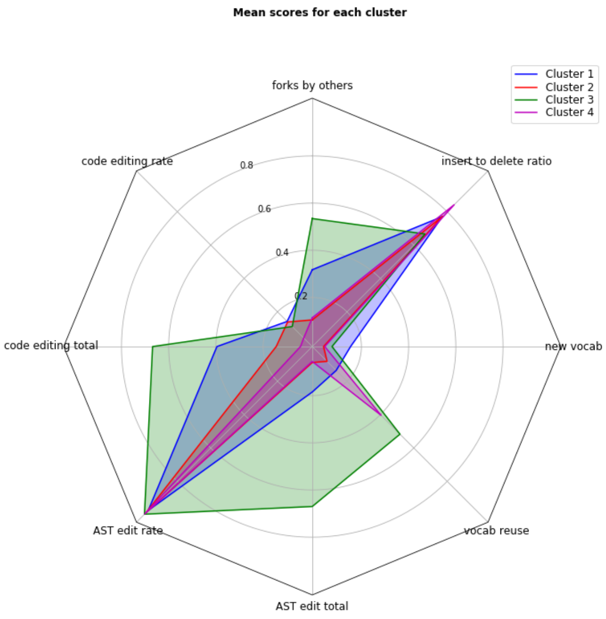


Figure 5: Mean values for normalised usage metrics for each cluster

Metric	**Cl 1**	**Cl 2**	**Cl 3**	**Cl 4**
forks by others	+	-	+	-
code edit rate		+		-
code edit total	+	-	+	-
AST edit rate				
AST edit total	+	-	+	-
vocab reuse	-	-	+	+
new vocab	+	-		
insert to delete ratio			-	+

Table 1: Correlations between clusters and metrics at p <0.05, with positive correlations shown by + and negative by -

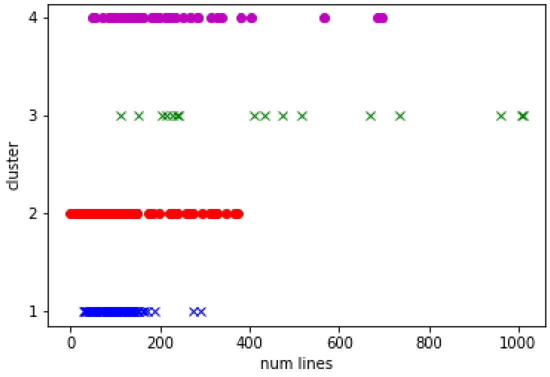


Figure 6: Lengths of the programs in each cluster

4 RESULTS

4.1 Clustering and code behaviours

Fig 4 shows the results of using the K-Means++ algorithm to cluster the coding metrics extracted from the descendants of 304 demonstrator programs using k=4. The clustering achieved a silhouette score of 0.39, demonstrating a successful clustering with a moderate density of clusters (scores closer to 1 represent high-density clusters).

Fig 5 shows the mean value for each cluster and metric. We normalised the metrics for easier comparison. While some metrics do not display much variance between the clusters, there are some apparent differences for others.

Following the approach used in [13], to characterise each cluster, we correlated the clusters (dummy coded) with the metrics using a point biserial correlation. Table 1 shows the results for significant positive and negative correlations. From this we can summarise the clusters in the following way: Cluster 1 has high overall change to code, with a favouring of picking up new vocab instead of reusing functions from the demonstrator program. Cluster 2 shows low overall change, both functionally and with the inclusion of new and seen vocab, however, these small changes were done quickly. Cluster 3 shows a high overall change, reusing the functions seen in the demonstrator code. Finally, Cluster 4 also sees high reuse though it combines this with small changes to the code in general. Clusters 1 and 3 showed high forking by other users, unlike Clusters 2 and 4.

4.2 Program authors and program length

We identified the three most prolific authors creating demonstrator code, all having either generated projects for multiple taught courses, or large numbers of examples for a specific research project. Instructor A conducted a large creative computing MOOC, as well as a similarly themed summer school focusing on the teaching of creative computing using a STEAM pedagogy. Instructor B used the platform to teach a different MOOC, as well as using it for in-lesson teaching at a university. Instructor C made several examples for teaching and encouraging use of a particular javascript library aimed at end-user machine learning, often in a creative context. The final set contains all other contributors; these may be students or other creators using the platform.

Performing a Pearson's chi-squared test, we found that clusters are significantly different from each other in respect to the four author categories (Instructor A, Instructor B, Instructor C, Other) (chi-square = 242.856, $p<0.05$). Cluster 1 has a higher proportion of programs authored by Instructor A; Cluster 3 is over-represented by Instructor C, and Cluster 4 has a larger than normal amount of documents from non-instructors (students or other contributors).

We conducted a Kruskal-Wallis test to examine the variation in program length between clusters, in terms of the number of lines of code. We found significant differences (Chi-square = 78.42, $p < 0.05$, df = 5) among the groups, indicating that the length of the program might be associated the types of coding behaviours seen in students. Fig 6 shows that Clusters 1 and 2 tend to have shorter programs, whereas Cluster 3 contains some much larger ones.

5 DISCUSSION

5.1 Coding Behaviours

Some pedagogical approaches used in creative computing encourage active learning through exploration and as such, characterising the process through which students develop their code is a promising investigative approach. Following the cluster analysis, our results demonstrate that students show several distinctly different types of coding behaviour.

We see from analysis of Cluster 1 that the demo programs were short and often authored by Instructor A. Students made large changes to the code, both in terms of edit operations and changes to the structure of the code, including introducing new vocabulary. Cluster 3 shows a similar pattern, except students tended to reuse vocabulary, as opposed to expanding it. This small cluster also contains all the programs from a specific instructor, and the programs are long and developed examples for a specific software library and research project.

Clusters 1 and 3 also showed high forking by others, meaning that people on average tended to go back and re-fork the code multiple times. But the higher incidence of new vocabulary in Cluster 1 indicates development towards a different application than the original whereas forks from Cluster 3 tended to re-use the same vocabulary, suggesting a more limited set of applications, closer to the originals. This appears to be inline with expectations, given the intention of some of the programs in the latter set.

Cluster 4 contains a disproportionate amount of programs by non-instructors where users comparatively edit the documents less. It seems possible that authors of non-instructor code have not designed it and have not used it for teaching purposes, so it is not surprising that its ongoing usage patterns are different to those for code that was designed for teaching. It would be interesting to investigate further why this code appears to be so popular.

Cluster 2, the largest cluster, has the most similar proportions of authors to those in the dataset as a whole, i.e. most authors seen in the other clusters are represented. Cluster 2 is characterised by limited amounts of code editing. The code was certainly used by students, as it was forked multiple times, but they did not edit it very much.

A correct interpretation of the limited editing that we observed in Cluster 2 depends on information about the context in which the code editing took place. Sometimes, instructors set simple tasks which only require a small amount of editing. For example, the instructor might show a technique to the students, ask them to experiment with the parameters in a demo program quickly, then move onto another technique. Further examination of the example programs is needed to establish the context. It is interesting that this cluster contains demo code from all the instructors, indicating that the limited code editing behaviour is likely neither lesson context nor instructor specific. Perhaps these demonstrator programs were just too uninspiring! Further work is required to explain this interesting variation.

6 CONCLUSION

Over the last five years, the authors have developed and delivered many arts and music-based computer science courses both online and in-person. We have developed and used a browser-based programming platform to support the delivery of our teaching, and it enables us to analyse how students develop code in our lessons. Many of our students begin by forking an existing program on the platform, and these demo programs are either explicitly designed course materials made by an instructor or created by other students using the platform.

Having some insight into how creative computing students engage with our demonstrator code is useful to creative computing tutors as it allows us to check if students are expressing the coding behaviours expected by the lesson. We can also uncover potentially unknown coding behaviours within a student population that can help inform tutors when giving feedback or designing new sessions. Knowing how students work on their code can aid the designing of new material and the refinement of pedagogical approaches. Being able to examine coding behaviours across large cohorts of students is especially useful as more of our courses involve online and distance learning, and we need effective, scalable ways to evaluate teaching which involves creative, open-ended tasks [23].

As such, we have presented a battery of metrics aimed at highlighting coding behaviours associated with learning creative computing concepts. We have analysed how the coding behaviours we have observed are indicative of different ways students approach learning to program and how we can connect these coding behaviours back to the context in which they occurred. We think the observations we have presented, and the approach for clustering coding behaviours represent valuable contributions to the learning analytics community and computer science educators more generally.

7 ACKNOWLEDGEMENTS

The work reported in this paper was supported by the Arts and Humanities Research Council under grant number AH/R002657/1, and the Higher Education Funding Council England Catalyst Scheme under grant number PK31.

REFERENCES

[1] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. 2006. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems - CHI '06*. ACM Press, Montr&#233;al, Qu&#233;bec, Canada, 231. https://doi.org/10.1145/1124772.1124808

[2] Matthew Berland, Taylor Martin, Tom Benton, Carmen Petrick Smith, and Don Davis. 2013. Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *Journal of the Learning Sciences* 22, 4 (Oct. 2013), 564–599. https://doi.org/10.1080/10508406.2013.836655

[3] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. 2014. Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming. *Journal of the Learning Sciences* 23, 4 (2014), 561–599. https://doi.org/10.1080/10508406.2014.954750

[4] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*. ACM Press, Atlanta, Georgia, USA, 513. https://doi.org/10.1145/1753326.1753402

[5] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Writing Code to Prototype, Ideate, and Discover. *IEEE Software* 26, 5 (Sept. 2009), 18–24. https://doi.org/10.1109/MS.2009.147

[6] Sayamindu Dasgupta, William Hale, Andrés Monroy-Hernández, and Benjamin Mako Hill. 2016. Remixing as a Pathway to Computational Thinking. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*. ACM Press, San Francisco, California, USA, 1436–1447. https://doi.org/10.1145/2818048.2819984

[7] John Dewey. 1916. *Democracy and Education: An Introduction to Philisophy of Education*. Macmillan.

[8] Brian Dorn and Mark Guzdial. 2010. Learning on the job: characterizing the programming knowledge and learning strategies of web designers. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*. ACM Press, Atlanta, Georgia, USA, 703. https://doi.org/10.1145/1753326.1753430

[9] C. A. Ellis and S. J. Gibbs. 1989. Concurrency Control in Groupware Systems *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 399–407. https://doi.org/10.1145/67544.66963

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[11] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Transactions on Computer-Human Interaction* 22, 2, Article 7 (2015), 35 pages. https://doi.org/10.1145/2699751

[12] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas J. Guibas. 2013. Syntactic and Functional Variability of a Million Code Submissions in a Machine Learning MOOC. In *Proceedings of the Workshops at the 16th International Conference on Artificial Intelligence in Education AIED (CEUR Workshop Proceedings)*, Erin Walker and Chee-Kit Looi (Eds.), Vol. 1009. CEUR-WS.org.

[13] Blair Lehman, Sidney K. D'Mello, Amber Chauncey Strain, Melissa Gross, Allyson Dobbins, Patricia Wallace, Keith Millis, and Arthur C. Graesser. 2011. Inducing and Tracking Confusion with Contradictions during Critical Thinking and Scientific Reasoning. In *Artificial Intelligence in Education*, Gautam Biswas, Susan Bull, Judy Kay, and Antonija Mitrovic (Eds.). Vol. 6738. Springer Berlin Heidelberg, Berlin, Heidelberg, 171–178. https://doi.org/10.1007/978-3-642-21869-9_24

[14] Yanchi Liu, Zhongmou Li, Hui Xiong, Xuedong Gao, and Junjie Wu. 2010. Understanding of Internal Clustering Validation Measures. In *2010 IEEE International Conference on Data Mining*. IEEE, Sydney, Australia, 911–916. https://doi.org/10.1109/ICDM.2010.35

[15] Seymour Papert. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[17] Marian Petre and Alan F. Blackwell. 2007. Children as Unwitting End-User Programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. 239–242. https://doi.org/10.1109/VLHCC.2007.52 ISSN: 1943-6106.

[18] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, 153–160. https://doi.org/10.1145/2157136.2157182

[19] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (Dec. 2011), 703–732. https://doi.org/10.1007/s10664-010-9150-8

[20] Sherry Turkle and Seymour Papert. 1990. Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs: Journal of Women in Culture and Society* 16, 1 (1990), 128–157. https://doi.org/10.1086/494648

[21] Benjamin Xie and Hal Abelson. 2016. Skill progression in MIT app inventor. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Cambridge, United Kingdom, 213–217. https://doi.org/10.1109/VLHCC.2016.7739687

[22] Seungwon Yang, Carlotta Domeniconi, Matt Revelle, Mack Sweeney, Ben U. Gelman, Chris Beckley, and Aditya Johri. 2015. Uncovering Trajectories of Informal Learning in Large Online Communities of Creators. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale - L@S '15*. ACM Press, Vancouver, BC, Canada, 131–140. https://doi.org/10.1145/2724660.2724674

[23] Matthew John Yee-King, Mick Grierson, and Mark d'Inverno. 2017. Evidencing the value of inquiry based, constructionist learning for student coders. *International Journal of Engineering Pedagogy (iJEP)* 7, 3 (2017), 109–129.

[24] Hezheng Yin, Joseph Moghadam, and Armando Fox. 2015. Clustering Student Programming Assignments to Multiply Instructor Leverage. ACM, New York, NY, USA, 367–372. https://doi.org/10.1145/2724660.2728695