Extracting Features for Computational Thinking from Block-Based Code

Param Damle
University of Virginia
Charlottesville, Virginia, United States
psd9vgc@virginia.edu

Jo Watts
University of Virginia
Charlottesville, Virginia, United States
jbw3r@virginia.edu

Glen Bull
University of Virginia
Charlottesville, Virginia, United States
glb2b@virginia.edu

N. Rich Nguyen
University of Virginia
Charlottesville, Virginia, United States
nn4pj@virginia.edu

ABSTRACT

To support undertrained instructors in introductory computer science classes, we proposed an automated evaluator (autograder) for *Snap!*, a block-based programming language whose colorful visual interface is more beginner-friendly. Our approach is not only novel in working natively on a non-textual language but also in its assessment of the computational thinking (CT) reflected in the structure of a student's submission rather than the accuracy or run-time of its execution. This relies on assessing demonstrated knowledge of abstraction and iteration from an XML tree representation of a student's *Snap!* program. Approaches supported by literature involve clustering trees with similar structures together; however, methods such as path matching were too generalized and inadequate at reflecting specific CT elements. To this end, we explore how to tailor our feature extraction to capture such elements, including consecutive repetition and encapsulation of functional blocks. Unlike proprietary autograders, our approach integrates the academic community into the research and development of the optimal feature embedding of *Snap!* programs; thus, we present both successful and unsuccessful endeavors to inform replications of this work. We also highlight avenues for feature tuning and scalability of the autograding model to larger, more diverse classrooms.

CCS CONCEPTS

• **Applied computing** → *Education*; • **Computing methodologies** → *Classification and regression trees*; *Learning latent representations*; • **Social and professional topics** → **Computational thinking**; **Student assessment**.

KEYWORDS

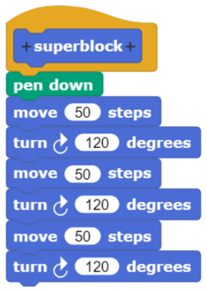
block-based programming, feature extraction, XML trees

ACM Reference Format:
Param Damle, Jo Watts, Glen Bull, and N. Rich Nguyen. 2023. Extracting Features for Computational Thinking from Block-Based Code. In *Proceedings of 29TH ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23)*. ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

Inadequate diversity in introductory computer science education, coupled with a shortage of technically trained instructors, presents significant barriers to learners in under-served areas [5]. Research suggests that incorporating media into computer science education can appeal to a broader audience and motivate students from such backgrounds to learn [4]. In line with this, we launched our MakeToLearn course in which novice coders learn programming using TuneScope, a music analysis and synthesis tool integrated into *Snap!* [1].

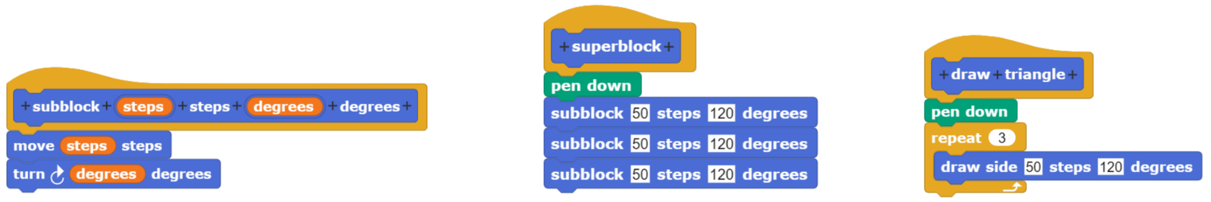
In this paper, we analyze how our rubric would evaluate an example program to understand the CT concepts we seek to capture in our features. A student's first attempt at a *Snap!* program that draws an equilateral triangle may look like Fig. 1, which contains repeated commands to draw a side and turn at the top level of code.



```
+superblock+
pen down
move 50 steps
turn ↻ 120 degrees
move 50 steps
turn ↻ 120 degrees
move 50 steps
turn ↻ 120 degrees
```

Figure (1) A naive approach to drawing a triangle

This is solved by encapsulating a "subblock" to handle an elementary function like drawing just one side, as shown in Figure 2. If a new instruction were needed before each side was drawn, this line would now only need to be added in one spot. This subblock can be used repetitively, but students would get algorithm points for placing it within an iteration. In this optimal version, modifying a parameter (e.g., side length) would only require an edit at one position.



```
+subblock+ steps +steps+ degrees +degrees+
move steps steps
turn ↻ degrees degrees
```

```
+superblock+
pen down
subblock 50 steps 120 degrees
subblock 50 steps 120 degrees
subblock 50 steps 120 degrees
```

```
+draw+triangle+
pen down
repeat 3
    draw side 50 steps 120 degrees
```

(a) Encapsulated subblock that draws one side **(b) Usage of subblocks in succession** **(c) Switch to a loop for efficiency**

Figure (2) A abstracted and efficient approach to drawing a triangle

This approach improves debugging efficiency and teaches programming concepts in a proactive, CT-conscious manner. The intuition behind the feature embedding, thus, lies in capturing the repetition and nested structure within each level of a student's code. Instructors provided personalized feedback on each assignment, addressing issues such as code redundancy or inadequate encapsulation. Although the data showed a highly positive effect on learning engagement and CT comprehension, the expansion of this course setup is limited by the availability of staff with adequate expertise and input [10].

To address these limitations, we're pursuing an autograder that provides personalized CT feedback to student submissions in a scalable manner. While autograders for programming assignments typically focus on run-time efficiency and output accuracy, effectively assessing students' CT skills requires evaluating coding best practices, including decomposition, abstraction, and algorithm design. While autograders are commonly used for text-based languages like Python, this work furthers efforts to extract meaningful CT features from originally block-based XML code data.

2 RELATED WORK

This model's potential pedagogical applications were previously presented to an audience of computer science educators by Damle et al. [2]; however, this report focuses on the techniques of data discovery and feature engineering involved in the process.

Previous studies have demonstrated that textual code can be effectively embedded in low-dimensional space with minimal training data [6–8, 11]. However, this relied on moving the model's cursor to positions in text containing incorrect syntax, a type of error that's almost impossible with *Snap!*'s prefabricated blocks. Furthermore, the abstracted deep learning approach black-boxes the feature extraction from human comprehension, hampering our ability to inform our data transformation *a priori*.

The design of our model was influenced by Piech et al.'s work [8], which successfully grouped code based on similarity in a scalable manner. The similarity measure here relied on functional testing of code output; in contrast, TuneScope assignments produce multimedia artwork without a predefined "correct" answer. Therefore, our grouping problem differs by treating the CT reflected in the code style as our target variable.

A notably similar project was created by the DrScratch team [3], whose model works on block-based language *Scratch* and assesses a variety of CT criteria. Many of the more advanced items (e.g. Parellelization, Synchronization, Flow Control) lie outside the scope of introductory computer science, and there's no clear metrics or syllabus provided for how DrScratch assesses the criteria relevant to this study. In general, the proprietary and undocumented nature of alternatives in the space call for an academia-facing autograder for *Snap!* whose primary goal is to produce the best tool for instructors while contributing to tree-based data science research.

3 METHODS

At the University of Virginia, we designed a course centered around Computational Thinking (CT) using a block-based programming language, *Snap!*, to teach novice programmers computer science fundamentals. The course is taught as an elective in the education department and is split into two main parts: art and music. First, students use the platform to replicate famous artists' styles using code, and then incorporate CS fundamentals and computational thinking into composing melodies and chord progressions to create multi-tracked songs, fully controlling instrument choice, volume levels, panning, and tempo.

Throughout the course, 13 students were exposed to structured, personalized instructor feedback on code submissions for 8 assignments. Each review consists of integral scores between 0 and 2 for abstraction, algorithms, data representation, and documentation from each reviewer, as well as an in depth textual response from a single reviewer providing constructive feedback and examples to the student. The dataset for this study comprises of *Snap!* assignments (as XML files) written by students in our course who were new to coding, along with the numerical score vectors from each reviewer and a text field for the personalized review.

Fundamentally, our autograder would compare the structure of the XML representation of each program to structures that both demonstrate and fail to demonstrate CT. Following the literature, this requires a clustering model that will use measures of "similarity" that correspond to CT to simultaneously apply feedback to large groups of submissions. The model's ability to cluster similar programs relies on both features that effectively allow the emergence of encapsulation and repetition patterns and a clustering algorithm that groups feature vectors similar to the manual rubric. Therefore, we conduct a study with a scan of possible features and a survey of popular clustering models with an analysis of the observed improvement of cluster proximity.

3.1 Tree-Based Features

Since the underlying format for *Snap!* programs is an XML tree, where sequential children of a parent node represent the script nested within the parent code block, the first features we extracted were focused on the structure of a generalized tree (e.g. average

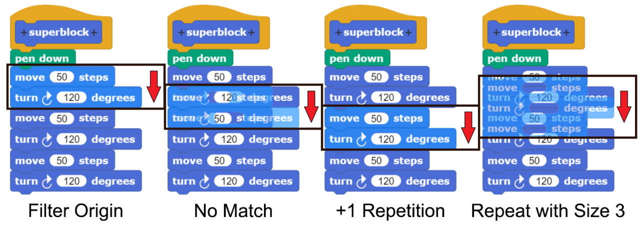


Figure (3) Sliding 1D filter for measuring repetition

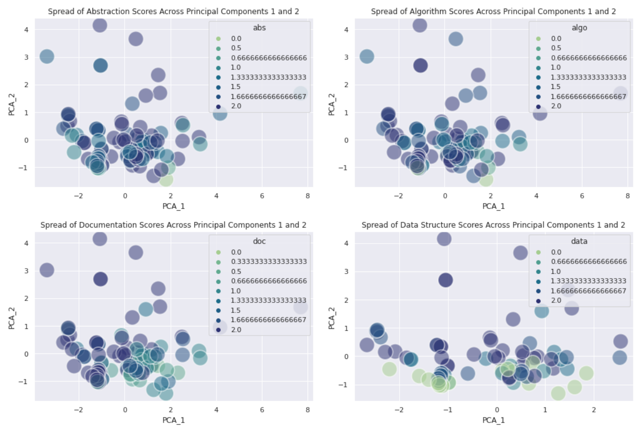


Figure (4) From top left, clockwise: student submissions plotted across highest PCA components colored by abstraction, algorithms, data representation, and documentation score

and maximum children per parent node, average and maximum depth of each child node). These were primarily numerical for ease of conceptual understanding and clustering familiarity, however methods exist to match higher-dimensional aspects of trees together, such as PathXP [9], which creates "profiles" of trees based on shared parent-child paths. Such an approach was not scalable since new path profiles would need to be compiled and measured across the entire dataset for every additional tree encountered.

3.2 Repetition

For measuring repetition, we invented a 1-dimensional convolution for *Snap!* programs, where a "filter" consisting of increasing sections of code blocks slides over the remainder of the code in Figure 3. All matching occurrences of the filter are tallied up to get a repetition score. In the example shown, a filter consisting of the blocks **move 50 steps** and **turn 120 degrees** is not matched one block below, but then matched 2 blocks below. After the tally of all such 2-block matches is collected, the process restarts with a filter size of 3, and so on, until the length of the script is reached as one big filter.

Block matches represent students repeating whole sections of code where they could have abstracted often-used functions and iterated often-repeated instructions. Note, just like in our other methods, the actual blocks (functions) in use matter less than how redundantly the student used them.

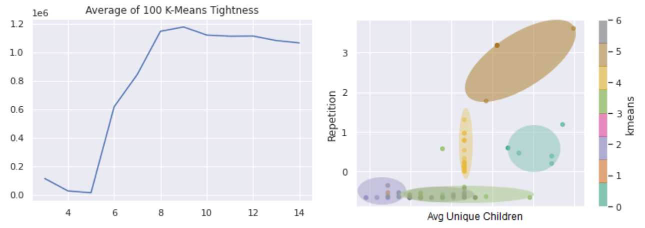


Figure (5) K-means clusters of student submissions

3.3 Feature Analysis

Our feature set focused on abstraction and algorithms, as data representation and documentation were not reflected in a tree structure. However, as shown in Figure 4, decomposing the feature vectors into the first two principal components and then comparing each point to its manual scoring showed greater spatial variance in the documentation and data representation than abstraction and algorithms as we had intended. This indicates an area of improvement for clever feature engineering, discussed in the Conclusion.

3.4 K-means Clustering

Despite the need for better feature embedding, we explore the options for clustering, starting with the most popular choice, K-means clustering. At first, proximity was an evenly-weighted L2 norm between feature vectors, and the inertia of a cluster assignment was the sum of squared error (SSE) across all points. As depicted in Figure 5, the optimal clustering had an "elbow" of 5 clusters for balancing low clusters with low inertia, and the corresponding clusters are plotted across features corresponding to our Repetition convolution and Average Unique Children per Parent Node.

The reasoning for plotting these two features is rooted in a grid search across feature importance. Every combination of weights between 0 and 1 (in 5 increments) was applied to each feature, where a higher weight means the feature impacted cluster proximity more. In Figure 6, the combinations of weights on every pair of features were plotted based on the resultant cluster inertia (averaged across multiple runs to minimize randomness). This grid search showed high weighting (1.0) on Repetition, and Average Unique Children features consistently produced lower cluster inertia, which should inform potential features moving forward.

3.5 Other Algorithms

For completeness, we compared a suite of other popular clustering algorithms to examine their potential in XML-extracted data. DBSCAN produced highly irregular inertia dependent on initialization and hyperparameter tuning, and always marked most of the points as noise, indicating data points were not spatially dense (as shown in Figure 7).

A Gaussian Mixture Model consistently separated the feature space into 2 clusters, generally distinguishing code with low and high Repetition (as shown in Figure 8).

Hierarchical clustering, shown in the dendrogram in Figure 9, also suggests 2 or 3 clusters, as the average height (variation between clusters) of the topmost (blue) tree is significantly higher than the other branches.

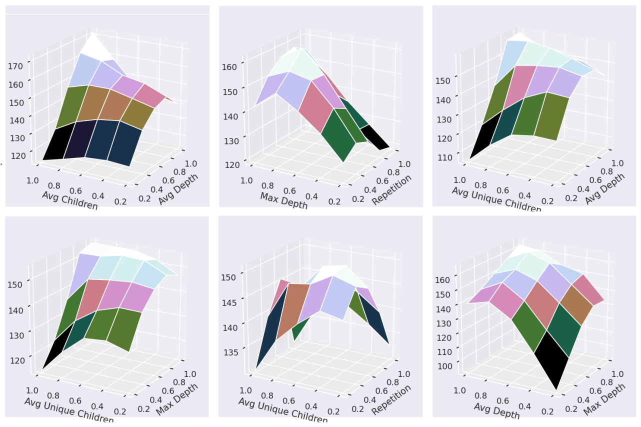


Figure (6) Grid search across inertia dependent on all combinations of feature weighting (select plots shown)

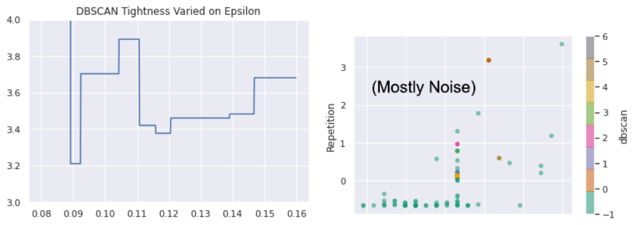


Figure (7) DBSCAN clusters of student submissions

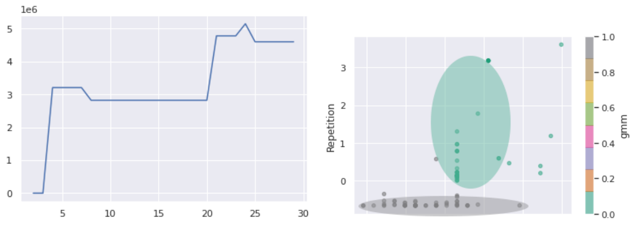


Figure (8) Gaussian clusters of student submissions

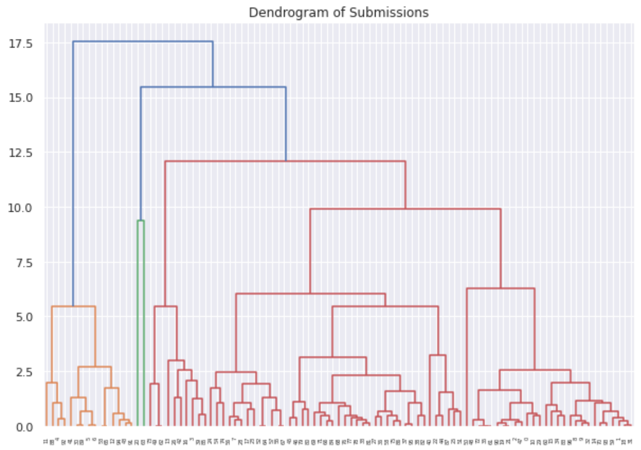


Figure (9) Hierarchical clusters of student submissions

4 EXPERIMENTAL RESULTS

Our benchmark for model performance would first require a significant overlap with numerical score vectors before we can pursue a model that provides human-like feedback.

Inertia, as described in Section 3.4, alone lacks a frame of reference for the error and would thus be an inaccurate measure of comparison between differently-weighted clustering runs. To this end, we introduce a metric for relative inertia, or error between each point and its cluster centroid, relative to the error to the closest alternative centroid (Equation 1). If each point was assigned to the appropriate cluster, the relative inertia would be less than 1; optimal clustering would achieve as close to 0 relative inertia as possible.

$$\mathbb{E}\left[\frac{(X-\boldsymbol{\mu}_k)^2}{\min_{j\neq k}(X-\boldsymbol{\mu}_j)^2}\right] \tag{1}$$

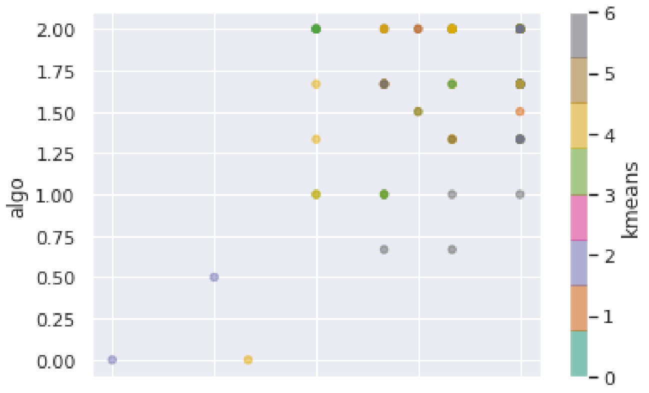
By clustering based on our embedded features and then measuring relative inertia across our manual scoring vectors, we hoped to assess which model best clustered according to our rubric before we applied text feedback. This format evaluates the training of the model on our engineered features (training data) using the verified CT scores provided by our instructors (validation/testing data). The relative inertia scores are compiled in Table 1.

Table (1) Relative Inertia of Various Clustering Algorithms

Methods	Relative Inertia
K-Means	3.2627
DBSCAN	27862
Gaussian	1.0822
HAC (2 clusters)	1.0481
HAC (3 clusters)	1.2021

Although Figure 10 shows a loose visual distinction in clustering for different manual CT scores in K-Means or Gaussian clustering individually, the near-1 relative inertia of Gaussian and Hierarchical models from Table 1 indicates the potential of a suite of models, each devoted to only one rubric item (such as a model only identifying repetitive algorithms).

Limitations. Drawbacks of our autograding model (compared to human graders) include lacking the ability to point out specific blocks the student created as areas of improvement, create personalized alternative coding solutions, and appreciate the artistic output of multimedia programs. With recent attention towards ChatGPT, we believe synthesized text tools, trained on personalized human feedback, could provide such tailored responses to new submissions. An alternate approach could include labeling generatively (by creating example code with common pitfalls, and then clustering actual submissions based on feature similarity). This might apply domain knowledge of how students typically think more easily [12]. As of today, over 45% of the class is comprised of female students, and the class boasts a 100% retention rate. In addition to the autograders, we would like to gather additional data from the creators of *Snap!*, who have graciously offered their insight and feedback on student submissions via the *Snap!* forum.



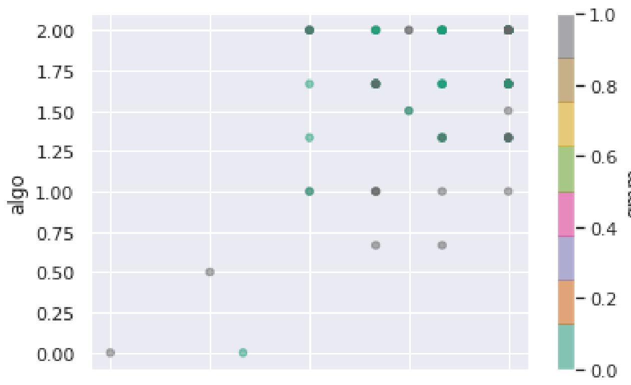


Figure (10) Plot of Algorithms score vs Abstractions score of each submission by K-Means (left) and Gaussian (right) clustering

5 CONCLUSION

This report introduces specific features that can be extracted from an XML representation of a *Snap!* program to capture patterns of encapsulation and repetition, as well as a convolution designed specifically for block-based code that measures redundancies in a student's script. Analyzing a variety of clustering algorithms indicates that rather than a density-based approach, models should consider only proximity between the submission at hand and other submissions that exemplify certain traits. Near-optimal relative inertia measures suggest the potential for general clustering algorithms to be used with feature vector representations of higher dimensional program data.

Our goal is to apply our model in the classroom and tune it as a proof-of-concept for our use case. This tuning mostly involves better-capturing domain knowledge about common student pitfalls in feature extraction. The crux of the problem moving forward will be finding this optimal embedding of block-based code that best distinguishes and clusters differently- and similarly-structured programs, respectively. Although prior work [8] describes automatically learning this embedding, a specific formula or algorithm would be useful in a variety of applications that capture patterns in tree-like data. Once functional, we seek to generalize the model to provide the desired scalability to multiple classrooms and, results permitting, clustering of tree-like data in other domains.

ACKNOWLEDGMENTS

Param Damle, an undergraduate student, was the proactive force behind the research, development, experimentation, and presentation of this project. He would like to thank Rich Nguyen for mentoring, advising, and encouraging him through the research process; Glen Bull, Jo Watts, and Rachel Gibson for allowing the Make To Learn class to serve as an ecosystem for this project's implementation; Annie Cao and Medhini Rachamallu for helping him provide the hours of manual feedback used as the project's dataset; Kartik Chugh for assisting in background research.

This collaboration between education researchers and machine learning engineers was funded by a 3Cavaliers grant for interdisciplinary projects and NSF STEM-C award #1842342.

REFERENCES

[1] Glen Bull, Rachel Gibson, Jo Watts, N. Rich Nguyen, and Luke Dahl. 2023. *TUNESCOPE Creating Digital Music in Snap!* https://www.learntechlib.org/p/221758

[2] Param Damle, Glen Bull, Jo Watts, and N. Rich Nguyen. 2023. Automated Structural Evaluation of Block-Based Coding Assignments *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1300. https://doi.org/10.1145/3545947.3576246

[3] DrScratch. 2014. Dr. Scratch, a Scratch projects analyzer. Retrieved 2019 from http://drscratch.org/

[4] Andrea Forte and Mark Guzdial. 2004. Computers for Communication, Not Calculation: Media as a Motivation and Context for Learning. In *Proceedings of the 37th Annual Hawai'i International Conference on System Sciences*. Big Island, Hawaii. https://doi.org/10.1109/HICSS.2004.1265259

[5] Joanna Goode. 2007. If you build teachers, will students come? The role of teachers in broadening computer science learning for urban youth. *Journal of Educational Computing Research* 36, 1 (2007), 65–88.

[6] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. AAAI Press, Honolulu, Hawaii, 930–937. https://doi.org/10.1609/aaai.v33i01.3301930

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

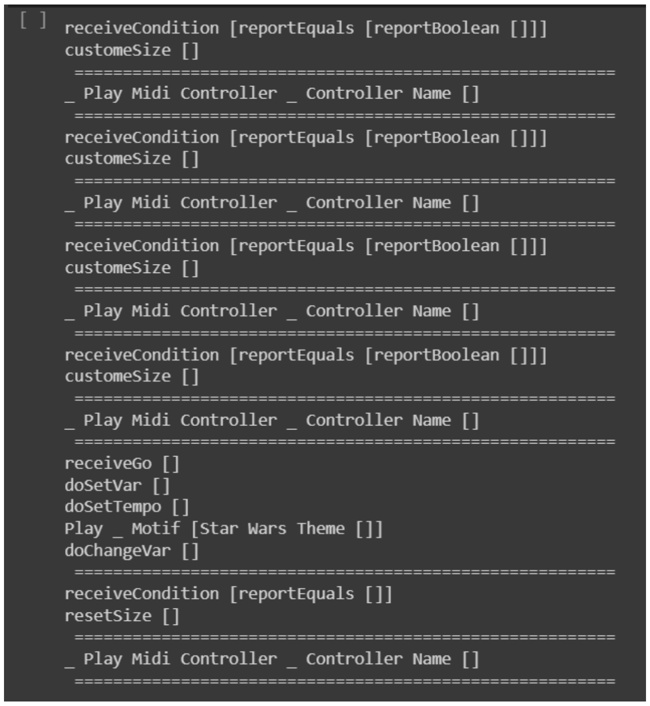
[8] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the 32nd International Conference on Machine Learning (W&CP, Vol. 37)*. JMLR, Lille, France, 248–253. https://doi.org/10.48550/arXiv.1505.05969

[9] Maciej Piernik. 2015. *Pattern-based clustering and classification of XML data*. Ph. D. Dissertation. Poznan University of Technology, Poznan, Poland. https://doi.org/10.13140/RG.2.1.3843.3763

[10] Patricia L Rogers. 2000. Barriers to adopting emerging technologies in education. *Journal of educational computing research* 22, 4 (2000), 455–472.

[11] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 15–26.

[12] Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. 2019. Zero Shot Learning for Code Education: Rubric Sampling with Deep Learning Inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. AAAI Press, Honolulu, Hawaii, 782–790. https://doi.org/10.1609/aaai.v33i01.3301782



```
[ ] receiveCondition [reportEquals [reportBoolean []]]
    customeSize []
     =====================================================
    _ Play Midi Controller _ Controller Name []
     =====================================================
    receiveCondition [reportEquals [reportBoolean []]]
    customeSize []
     =====================================================
    _ Play Midi Controller _ Controller Name []
     =====================================================
    receiveCondition [reportEquals [reportBoolean []]]
    customeSize []
     =====================================================
    _ Play Midi Controller _ Controller Name []
     =====================================================
    receiveCondition [reportEquals [reportBoolean []]]
    customeSize []
     =====================================================
    _ Play Midi Controller _ Controller Name []
     =====================================================
    receiveGo []
    doSetVar []
    doSetTempo []
    Play _ Motif [Star Wars Theme []]
    doChangeVar []
     =====================================================
    receiveCondition [reportEquals []]
    resetSize []
     =====================================================
    _ Play Midi Controller _ Controller Name []
     =====================================================
```

Figure (11) Compact representation of XML Snap! program

6 REPRODUCIBILITY

We've published a Google Colab Python Notebook that demonstrates how to parse *Snap!* programs for their structure, script, and metadata. The notebook is viewable online at https://colab.research.google.com/drive/1cdyqz0juK1hnQW4-mKbc2woiiMhpZUWt?usp=sharing. Actual student code and manual feedback is not provided, but interested persons can find publicly available programs on the *Snap!* website (https://snap.berkeley.edu/).

The main module that allows users to work with XML data in Python is `etree`. Since *Snap!* programs contain a lot of overhead and metadata, code is already in place to reduce the XML program to a custom `CodeBlock` object, which you can tune to fit your needs. For the purposes of measuring block redundancy in this project, we only kept the name of the function block. Figure 13 shows a print of a sample TuneScope program in this format, where text inside the square brackets [] represents nested code and the underscore _ represents an omitted parameter value.

The provided notebook also includes a framework for users to add their own feature extraction functions (provided that the resulting data format is one numerical feature vector per student submission) and run the analyses provided in this report, including clustering with a suite of models, grid search over feature importance, and (provided a properly formatted matrix of scores) plotting of clustered data on a 2D score plot.

Proper formatting of a score dataset would be one row per submission, with the following fields:

- **filename**: the name of the XML file corresponding to the project (omitting the `.xml` extension), located within the `base_path_xml` folder specified at the top of the notebook)
- **r#_CAT**: the reviewer score of this project, where # is the reviewer number (setup as 1–3 in the notebook) and **CAT** is the criteria category, which in the notebook is given as **abs**traction, **algo**rithms, **doc**umentation, and **data** representation.

Users can also include a text file containing the names of any code blocks that are common across many submissions or part of a library, which *Snap!* often lists as custom blocks when imported, so that they are not factored into the analysis.