Automated Classification of Visual, Interactive Programs Using Execution Traces

Wengran Wang
North Carolina State University
wwang33@ncsu.edu

Gordon Fraser
University of Passau
gordon.fraser@uni-passau.de

Tiffany Barnes
North Carolina State University
tmbarnes@ncsu.edu

Chris Martens
North Carolina State University
crmarten@ncsu.edu

Thomas Price
North Carolina State University
twprice@ncsu.edu

ABSTRACT

Offering students immediate, formative feedback when they are programming can increase students' learning outcomes and self-efficacy. However, visual and interactive programs include dynamic user input and visual outputs that change over time, making it difficult to automatically assess students' code with traditional functional tests to offer this feedback. In this work, we introduce Execution Trace Based Feature Engineering (ETF), a feature engineering approach that extracts sequential patterns from execution traces, which capture the runtime behavior of students' code. We evaluated ETF on 162 students' code snapshots from a Pong game assignment in an introductory programming course, on a challenging task to predict students' success on fine-grained rubrics. We found that ETF achieves an average F_1 score of 0.93 over 10 grading rubrics, which is 0.1–0.2 higher than a high-performing syntax-based code classification approach from prior work. These results show that ETF has strong potential to be used for code classification, to enable formative feedback for students' visual, interactive programs.

Keywords

execution traces, feature engineering, computer science education, code classification, formative assessment

1. INTRODUCTION

Real-time, formative feedback promotes students' learning gains and self-efficacy [7, 3, 12, 18]. To provide such formative feedback in real-time, CS instructors commonly write test cases, allowing students to run their code against these test cases when programming [9, 5, 6, 4]. However, visual, interactive programming projects, such as creating apps and games [16], include dynamic user interactions, and visual outputs that change over time, making it challenging to use test cases to assess these programs [14, 13, 20].

In contrast to test case-based approaches, data-driven methods allow instructors to offer formative feedback by grading a smaller set of programs instead of writing test cases [11, 21, 10, 22]. These methods start with transforming code into input vectors using **feature engineering**, typically by extracting syntax elements from an abstract syntax tree (AST), where nodes and their children correspond to specific code elements (e.g., if statements). However, when applying these syntax-based feature extraction techniques to classify programs, prior work showed mixed results, which are often not high enough to ensure the quality of student feedback [11, 1, 2]. Some prior work has used **execution traces** to classify students' sorting programs based on their specific strategies, and has shown that execution-trace-based classification achieved higher accuracies than a syntax-based classification approach [8]. However, no prior work has conducted feature extraction on the execution trace of visual, interactive programs, which include dynamic user interactions and various changing outputs. In this work, we extract features from execution traces that capture the runtime behavior of visual, interactive programs. We designed an execution trace-based feature engineering approach (ETF) to transform students' source code into feature vectors, for classification algorithms to build models based on rubric-based labels (e.g., the presence of a key-triggered movement). We evaluated ETF by classifying 162 students in-progress and submitted code snapshots. We found it to achieve high prediction performance with an average of 0.93 F_1 score over 10 grading rubric items, which is 0.1–0.2 higher than a high-performing syntax-based code classification approach. Our work has the following contributions: 1) We designed and implemented a novel, execution trace-based feature engineering (ETF) approach to extract temporal patterns in students' visual, interactive programs; 2) We evaluated the ETF approach on students' code snapshots for a widely-used, representative visual, interactive program assignment.

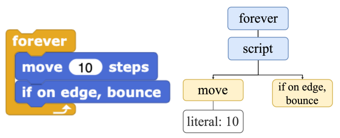
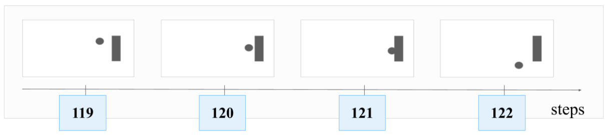


Figure 1: A horizontal n(2)-Gram (in yellow) and a vertical n(2)-Gram (in blue).

2. RELATED WORK

Syntax-based approaches extract patterns inside a code AST, use the presence or absence of a feature, or the count of the feature to generate input vectors. As an example,



a) Automated Program Execution

clock time	step	ball		
		position	direction	TouchSprite
1.974s	119	x:194, y:14	146	{ }
1.992s	120	x:200, y:6	146	{ }
2.019s	121	x:205, y:-3	214	{paddle}
2.024s	122	x:183, y:-35	214	{ }

b) Dumping Execution Traces

Figure 2: Step 1: Generating Execution Traces.

we explain a recently-applied AST n-Gram feature extraction approach [17, 2] by making an analogy to Natural Language Processing (NLP): In NLP, an n-Gram with $n = 1$ is a *1*-Gram feature taken from each word; and an n-Gram feature takes a continuous sequence of n words to extract relationship between words (e.g., orders); in an AST, *1*-Gram features represent each *node* in students' code. And to extract structural relationships, Akram et al. [2] designed the use of n-Grams to represent n-length sequences of code, where a ***vertical n-Gram*** is created by a depth-first search of leaves; a ***horizontal n-Gram*** is created by a breadth-first search of all direct children of each AST nodes (e.g., in Figure 1) [2]. *1*-Gram and n-Gram-based AST feature extraction approaches have both been applied for student code classification tasks. Compared to *1*-Gram feature extractions, prior work has shown that n-Grams provide more predictive features for code analysis. For example, Akram et al. used n-Grams with n ranging from 1 to 4 to extract features, and used a Gaussian Process model to infer scores on 642 students' code pieces, in a block-based programming environment. This achieved an R-square of 0.94, higher than the 0.88 achieved by the baseline *1*-Gram approach [2, 1].

On the other hand, recent work by Paaßen et al. used **execution-trace-based** distance measures to classify programs into different strategies (e.g., bubble sort v.s. Insertion sort), and found that execution- race-based classification achieving 90% accuracy, higher than the 80% accuracy achieved by syntax-based approaches [8]. This shows the potential of using execution traces to classify students' programming code.

3. THE ETF APPROACH

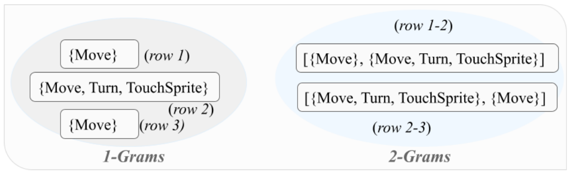
ETF starts from collecting a set of students' programming code, along with a class label for each piece of code, (i.e., positive or negative). ETF is designed for programs that have the following properties: 1) respond to **dynamic user inputs** (e.g., mouse, keyboard). 2) has **object-specific program states**, corresponding to visual output on the screen. 3) Program behaviors can be **a function of time**; and can also **change over time**.

An Example Assignment. As an example, consider the Pong assignment, which consists of a paddle sprite and a ball sprite. The ball moves around the stage [19], and a player can use the keyboard to control the up and down movement of the paddle to catch the running ball. If the paddle catches the ball, the player score increases; but if the paddle misses the ball and the ball hits the back wall, the game ends. ETF conducts feature engineering on such visual, interactive programs in four steps, described below.

step window	ball				
	Move?		Turn?	TouchSprite?	
119 - 120	✓	{right, down}	✗	✗	
120 - 121	✓	{right, down}	✓	✓	{paddle}
121 - 122	✓	{left, down}	✗	✗	

Summary Trace:
row 1: {Move({right, down})}
row 2: {Move({right, down}), Turn, TouchSprite({paddle})}
row 3: {Move({left, down})}

a) Step 2: Generating Summary Trace


{Move} (row 1)
{Move, Turn, TouchSprite} (row 2)
{Move} (row 3)
1-Grams
(row 1-2)
[{Move}, {Move, Turn, TouchSprite}]
[{Move, Turn, TouchSprite}, {Move}]
(row 2-3)
2-Grams

b) From Step 3: 1-Grams and 2-Grams

Figure 3: Step 2&3: summarizing traces & generating features.

3.1 Step 1: Generating Execution Traces

Visual, interactive programs include program states that can be represented as ***properties*** that change over time. For example, in Snap*!*, these properties can include: 1) ***Time***: how much milliseconds has passed from the start of program execution; 2) *inputs*: including `KeysDown` (which key is pressing); `MouseDown` (if mouse pressing); `MouseX` and `MouseY` (x, y positions of mouse) 3) *global variables*: the names (`Var.Name` and values `Var.Value`) of global variables; 4) *sprite-specific properties*: properties that are related to specific sprites, such as (`x`, `y`) (x, y coordinates); `dir` (directions); `TouchSprite` (which sprites the current sprite is touching); `TouchEdge` (which stage edge the sprite is touching); `size` (sprite size); `OffStage` (whether the sprite is moved out of the stage); the names (`Var.Name`) and values `Var.Value` local variables. The dumped execution trace tables use sprite names to label sprite-specific properties (e.g., to distinguish `ball.x` from `paddle.x`[1]).

Systems such as Snap*!* and SCRATCH make use of ***step functions*** to update the above properties based on the current properties and the current user inputs [14, 19]. We instrumented the step function in Snap*!* with a trace logging tool, so that with each Step, it adds a row in an ***execution trace table*** with the properties listed above, and dumps the trace table at the end of the execution. Figure 2 gives an example of a part of the *execution trace table*, in which one row logs one discrete Step created by the step function, with each entry maps to a *property* (i.e., a concrete program state).

3.2 Step 2: Summarizing Traces

ETF algorithm scans through the execution trace table in a sliding window of multiple steps (default as 2), apply a ***Trace Abstraction Function (TAF)***. The TAF looks for properties based on ***candidate properties***, and only returns properties that were found in the sliding window as a *summarized property set*. A *candidate property* can be an ***abstract property***, describing the changes between steps in the execution trace, such as movement and variable change; A *candidate property* can also be an original trace *property* which were already recorded in the execution trace. In each sliding window, the TAF function returns a *summarized property set* that includes all found properties, for example, in the step window 120-121 of Figure 3, all candidate properties were found because there is a change in po-

[1]ETF uses these properties to summarize trace and generate features (Section 3.3). To allow comparison across students, sprites need to have consistent labels across student programs.

sition (`Move`), a change in direction (`Turn`), and a non-empty `TouchSprite` column in Step 121. This creates a *summarized property set* {`Move`, `Turn`, `TouchSprite`}, shown in Row 2 of the *summary trace* (Figure 3). In addition, TAF's candidate properties also include possible types of ***parameters***, which describe detailed information of the property.

For Pong, ETF used 9 types of ***candidate properties***. Except 2 program state properties: (`KeysDown` and `ChangeVar`), the rest 7 are sprite-specific properties and are labeled with the sprite names (e.g., `Paddle.Move`). Among the 9 *candidate properties*, 4 were original trace *properties*, directly returned when the corresponding property in the last step of the sliding window is non-empty: `KeysDown`, `TouchSprite`, `TouchEdge`, and `OffStage`, using the same parameters with the corresponding execution trace entry at the last step of the sliding window, explained in Section 3.1. Others are 5 *abstract properties*, that only checks if a property changes between the first and last step of the sliding window (and if has middle, omit those middle ones). 1) `Move`$\langle\leftarrow,\rightarrow,\uparrow,\downarrow\rangle$. Returned when a sprite position changes. Its parameters are the direction toward which the sprite moves. 2) `Turn`. Returned when a sprite changes direction. 3) `ChangeSize`⟨`+, -`⟩. Returned when the Sprite changes its size to bigger (+) or smaller (-). 4) `ChangeVar`⟨`variable names`⟨`+, -`⟩⟩. Returned when a global variable's value has been changed to bigger (+) or smaller (-). 5) `ChangeLocalVar`⟨`variable names` ⟨`+, -`⟩⟩. Returned when a local variable's value has been changed to bigger (+) or smaller (-).

3.3 Step 3: Generating n-Gram-based Features

ETF next transforms *summary trace* created by Step 2 into a set of features using n-Gram-based approach, where an n-Gram takes a contiguous sequence of n items in data (Figure 3). ETF extracts features of 4 types: 1) ***1*-Grams**, extracting **simultaneous** behaviors, taken from each row of the summary trace. 2) ***2*-Grams**, connecting adjacent *1*-Grams sequentially; 3) **Power Sets.** We extract n-Grams of not only the full property set in each row of the *summary trace*, but also of subsets of the property set, such as the 2-set of just `Move` and `Turn` from t *1*-Gram {`Move`, `Turn`}. When constructing power sets for *2*-Grams, we apply the power set on the types of properties that are possible in this *2*-Gram. 4) For all the n-Grams extracted above, we keep both ***non-parameterized n-Grams***, where we do not record the parameters, as well as ***parameterized n-Grams***, where each property would include its parameters when they were logged in the summary trace. Next, ETF collects distinct features from all students' feature sets as the ***full feature set***, which consists of distinct features from all students.

3.4 Step 4: Filtering Features

Merging duplicate features and removing rare features. Based on the *full feature set* generated from Step 3, if features have the exact same distribution among programs, the ETF algorithm then merges these features as one feature; and it calculates the support of each feature based on the proportion of student programs that include this feature, and remove features that have support lower than a certain threshold, determined by a hyperparameter tuning process, described in Section 4.2.

Generating $\vec{x}$ vectors. After generating, merging duplicates, and removing rare features, we use the resulting feature set as the independent variables, and for each student program, we use 1 as representing the presence of a feature in the student program (i.e. the n-Gram appeared at least once in their abstracted execution trace), 0 as the absence of the feature, and generate 0-1 digitized $\vec{x}$ vector for each student's code snapshot, which is used as vector input for a classification model.

4. EVALUATION

We investigate our research question: **How accurately does ETF perform rubric-based code classification of students in-progress and submitted code, and how does this compare to syntax-based approaches?**. We first a) compare performance of ETF features and syntax-based features **across models**; and next b) compare ETF features and syntax-based features **across rubrics** on a fixed model. Our analysis of a) and b) follows the same procedure, where we started by generating ETF and syntax-based features separately (Section 4.1). We next performed the same feature filtering, training and evaluation procedure on the features we created (Section 4.2).

Dataset. We evaluated ETF on 42 students' 162 code snapshots for a Pong game assignment, sampling student code snapshots at 10 minutes (42), 20 minutes (40) and 30 minutes (38) of work, as well as their final submissions (42), on 10 target evaluation items: key_up/down, upper/lower_bound, space_start, edge_bounce, paddle_bounce, paddle_score, reset_score, reset_ball. A detailed description and the prevalence of the data can be found in our prior work [19].

4.1 Generating Features

Generating ETF features. We used the procedure described in the Step 1 – 3 (Section 3.1–3.3) of the ETF approach to generate ETF features. We first automatically run the program based on inputs. To ensure coverage, we used the same inputs (up/down arrow key, follow/evade ball) as our in prior work [19], defined using SnapCheck. For each program snapshot, we re-executed the program 5 times. Each run of student programs generated one execution table.

Generating AST *n*-Gram and *1*-Gram Features. We first compare ETF it with a representative, syntax-based feature extraction approach that has performed well in prior evaluations by using the AST n-Gram feature extraction approach [2]. Similar to Akram et al.'s work, we extracted all n-Grams from all ASTs, using $n = 1$ to 5 for *vertical n-Grams*, and $n = 1$ to 4 for *horizontal n-Grams* (explained in Section 2). Similar to many AST feature extraction approaches [10, 11, 22], we used a single label for all literals (`literal`).

4.2 Feature Filtering & Evaluation

We applied the same feature filtering and evaluation to the ETF, AST n-Gram, and AST *1*-Gram features:

Feature Filtering. For fairness of comparison, after collecting features, we used the Step 4 from the ETF algorithm to filter features for all ETF, AST n-Gram, and AST *1*-Gram features. For each type of the three features, we

Table 1: F1 scores of AST 1-Grams, AST n-Grams, and ETF Features, over different models.

	AST 1-Grams	AST n-Grams	ETF Features
Logistic Regression	0.771	0.779	0.932
AdaBoost	0.78	0.78	0.922
Random Forest	0.763	0.773	0.926
MLP	0.764	0.771	0.908
Gaussian Process	0.739	0.728	0.923
SVM	0.759	0.771	0.93

started by using ETF to automatically merge duplicate features (Step 4.a), and remove features that have support smaller than a certain threshold in the training set (Step 4.b). The threshold is set as a hyperparameter (tuned as described below).

Classification Models. To ensure that our comparison was not model-specific, we used 6 different models on the feature set: Logistic Regression, AdaBoost, Random Forest, Multi-layer perceptron (MLP), Gaussian Process, and SVM. Among them, the Gaussian Process model with an RBF kernel was also employed by Akram et al., and has shown to be the best performing model in the rubric-based performance inference task that they have applied [2].

Training & Evaluation. We employed 10-fold cross-validation to evaluate how accurately these different features predict the rubric-based performance. Within each round of cross-validation, we used another 2-fold cross-validation to tune the hyperparameters (i.e. nested cross-validation [15]). For all models, we included a minimum feature support threshold hyperparameter, T, below which we exclude the feature (e.g. ETF feature or AST n-Gram feature) from the final feature set, with the minimum support threshold as a hyperparameter, tuned based on 5 values: $\{0, 5\%, 10\%, 15\%, 20\%\}$. Additionally, since different feature extraction approaches may perform best with different values of model-specific hyperparameters, we also tuned hyperparameters for each classification models, based on the following values: *Logistic regression*: with penalty in {L1, L2}; *Random Forest*: with n_estimaters (i.e., number of trees in the forest) in $\{100, 200, 300, 400, 500\}$; *AdaBoost*: with learning rate in $\{0.01, 0.1, 1\}$; *MLP*: with learning rate in $\{0.001, 0.01, 0.1\}$; ' *SVM*: We used a linear kernel, with the regularization parameter (C) in $\{0.01, 0.1, 1, 10, 100\}$; Gaussian Process models optimize kernel hyperparameters during model fitting, we therefore did not tune hyperparameters for the Gaussian Process classifier. The values of the minimum feature support threshold and the model-specific hyperparameters were determined by their F_1 scores in the nested 2-fold cross-validation, based on a grid search on 5*#(model-specific hyperparameter values) possible types of hyperparameter combinations, during each round of the 10-fold cross-validation. Since many of our target labels are imbalanced, the accuracy score offers less information on how well our model performs in predicting target labels. We therefore use F_1 scores to tune hyperparameters. To ensure that data from a given student was not contained in *both* training and testing sets, all cross-validation splits were done on the 42 **students** (instead of on the 162 snapshots).

4.2.1 Results

Comparison Across Models. Using each of the 6 models described above, we predict students' rubric-based performance, calculated its F_1 score among the 162 students' data using 10-fold cross validation, creating one F_1 score for each rubric. We next averaged F_1 scores for each classifier across all rubrics, shown in Table 1. We saw that the AST n-Gram approach performed similar to the *1*-Gram approach (5 in 6 cases), and that ETF features generated F_1 scores that were consistently 0.14 to 0.2 higher than the AST n-Gram features, showing that all classifiers benefited from the execution-trace-based information extracted by the ETF features, with overall F_1 scores between 0.9 and 0.93. This result shows potential for us to make use of ETF features to correctly analyze students' current progress, which should enable automated, formative feedback in future work, to help a student who is stuck in the middle of programming.

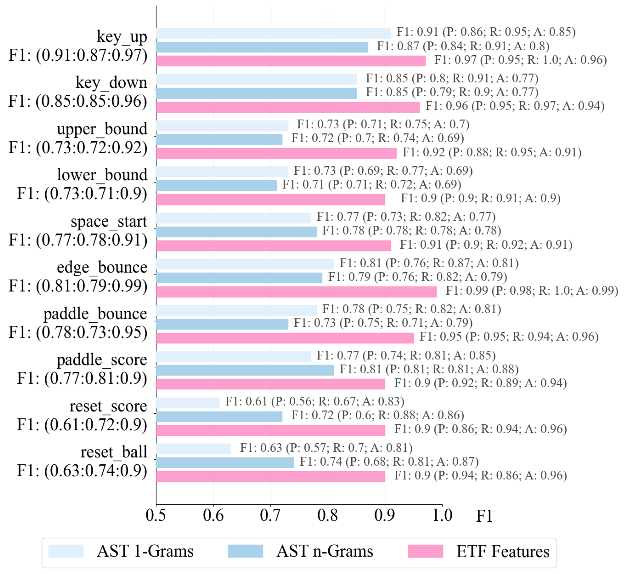


Figure 4: The F_1 score (F1), Precision (P), Recall (R) and Accuracy (A) of ETF, *n*-Gram, and *1*-Gram features on each rubric, using an SVM model. x-axis starts from F_1 = 0.5.

Performance Across Rubrics. We next investigate the performance of the three feature extraction approaches **across rubrics**. Since all model show similar trends, we select SVM and present performance on all rubrics in Figure 4. We found 1) The naive AST *1*-Gram features had relatively lower F_1 scores on rubrics that had less prevalence in data (e.g., *reset_score*, *reset_ball*); 2) Comparing to AST *1*-Gram, the AST n-Gram features produced higher F_1 scores for *paddle_score*, *reset_score*, *reset_ball*, showing that AST n-Gram extracted more useful feature for these three rubric items. However, the ETF features performed relatively well across all rubrics, with its F_1 scores ranging from 0.9 to 0.99, showing that ETF features have strong potential to enable formative feedback on a variety of fine-grained, specific rubrics.

In conclusion, we presented a novel, effective approach that extracted useful features from execution traces (ETF), leading to high predictive accuracy. Our results show strong potential for using ETF to monitor student progress and offer automated, formative feedback.

5. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1917885.

References

[1] B. Akram et al. Assessment of students' computer science focal knowledge, skills, and abilities in game-based learning environments. 2019.

[2] B. Akram, W. Min, E. Wiebe, A. Navied, B. Mott, K. E. Boyer, J. Lester, et al. Automated assessment of computer science competencies from student programs with gaussian process regression. In *Proceedings of the 2020 Educational Data Mining Conference*, 2020.

[3] A. T. Corbett and J. R. Anderson. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 245–252, 2001.

[4] S. H. Edwards and K. P. Murali. Codeworkout: short programming exercises with built-in data collection. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 188–193, 2017.

[5] D. Hovemeyer and J. Spacco. Cloudcoder: a web-based programming exercise system. *Journal of Computing Sciences in Colleges*, 28(3):30–30, 2013.

[6] A. N. Kumar. Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors. *Technology, Instruction, Cognition and Learning (TICL) Journal*, 4(1), 2006.

[7] S. Marwan, G. Gao, S. Fisk, T. W. Price, and T. Barnes. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 194–203, 2020.

[8] B. Paaßen, J. Jensen, and B. Hammer. Execution traces as a powerful data representation for intelligent tutoring systems for programming. *International Educational Data Mining Society*, 2016.

[9] A. Patil. Automatic grading of programming assignments. 2010.

[10] T. W. Price, Y. Dong, and T. Barnes. Generating data-driven hints for open-ended programming. *International Educational Data Mining Society*, 2016.

[11] Y. Shi, K. Shah, W. Wang, S. Marwan, P. Penmetsa, and T. Price. Toward semi-automatic misconception discovery using code embeddings. In *The 11th International Conference on Learning Analytics and Knowledge (LAK 21)*, 2021.

[12] V. J. Shute. Focus on Formative Feedback. *Review of Educational Research*, 78(1):153–189, 2008.

[13] A. Stahlbauer, C. Frädrich, and G. Fraser. Verified from scratch: Program analysis for learners' programs. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 150–162. IEEE, 2020.

[14] A. Stahlbauer, M. Kreis, and G. Fraser. Testing scratch programs automatically. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 165–175, 2019.

[15] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974.

[16] W. Wang, A. Kwatra, J. Skripchuk, N. Gomes, A. Milliken, C. Martens, T. Barnes, and T. Price. Novices' learning barriers when using code examples in open-ended programming. ITiCSE '21. Association for Computing Machinery, 2021.

[17] W. Wang, Y. Rao, Y. Shi, A. Milliken, C. Martens, T. Barnes, and T. W. Price. Comparing feature engineering approaches to predict complex programming behaviors. 2020.

[18] W. Wang, Y. Rao, R. Zhi, S. Marwan, G. Gao, and T. W. Price. Step tutor: Supporting students through step-by-step example-based feedback. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 391–397, 2020.

[19] W. Wang, C. Zhang, A. Stahlbauer, G. Fraser, and T. Price. Snapcheck: Automated testing for snap programs. ITiCSE '21. Association for Computing Machinery, 2021.

[20] W. Wang, R. Zhi, A. Milliken, N. Lytle, and T. W. Price. Crescendo : Engaging Students to Self-Paced Programming Practices. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2020.

[21] M. Wu, M. Mosse, N. Goodman, and C. Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 782–790, 2019.

[22] R. Zhi, S. Marwan, Y. Dong, N. Lytle, T. W. Price, and T. Barnes. Toward Data-Driven Example Feedback for Novice Programming. *Proceedings of the International Conference on Educational Data Mining*, pages 218–227, 2019.