Execution-Trace-Based Feature Engineering To Enable Formative Feedback on Visual, Interactive Programs

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.

1. INTRODUCTION

Real-time, formative feedback promotes students' learning gains and self-efficacy [20, 8, 33, 40], for it allows students to judge how well they are meeting assignment requirements, and make modifications to their code, both as they develop it, and before submitting it. 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 [24, 12, 16, 10]. However, visual, interactive programming projects, such as creating apps and games [38] include dynamic user interactions, and visual outputs that change over time, making it challenging to use test cases to assess these programs [35, 34].

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 [32, 43, 27, 48]. These data-driven 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 AST feature extraction techniques to classify programs based on fine-grained assignment rubrics, prior work showed mixed results, which are often not high enough to ensure the quality of student feedback [32, 1, 2].

A known key limitation of such AST-based feature engineering approaches is that code is first and foremost "executable", and that its execution traces include dynamic data about the program functionality, which cannot be directly captured by the syntax-based features [23, 3]. 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 [23]. 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 explore extracting useful 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 (Section 3); 2) We evaluated the ETF approach on students' code snapshots for a widely-used, representative visual, interactive program assignment (Section 4).

2. RELATED WORK

Students can easily get stuck during programming [14], and may need help in order to proceed. Such timely feedback that is provided in the middle of programming is called **"formative feedback"**. Unlike summative feedback, offered after a student completing a programming task, formative feedback aims to monitor students' on-going learning process and provide timely assessment and suggestions [11]. Receiving such formative feedback has been shown to improve students' learning outcomes [8] and self-efficacy [20]. In programming education, such formative feedback is built on an accurate assessment of students' current progress based on their code, which can take the form of analyzing students' progress against instructor rubrics [32]. With the growing size of CS classrooms [6], such assessment is challenging to offer manually.

To enable formative feedback, many learning systems make use of functional tests; some used syntax-based approached to generate hints [26, 50], other researched data-driven code classification. We first review prior work on test-case-based assessment to understand its benefits and limitations. Next, we describe prior work on data-driven code classification, which inspired our work.

1) **Test-case-based Automated Assessment.** Many learning systems offer test-case-based automated assessments, where students are allowed to submit their solutions multiple times. With every submission, the system checks students' solutions through a set of test cases and sends the report to the students (e.g., [24, 12, 16, 10]). However, visual and interactive programs include dynamic user interactions and program properties that change over time [35, 34]. Prior work that has applied functional tests to analyze these programs identified shared challenges for instructors to author such test cases, which requires temporal logic-based specifications of program behaviors and time bounds, using a domain-specific language [35, 41].

2) **Data-driven Student Code Classification.** Prior work has made use of data-driven methods to classify student programs for enabling formative feedback. These methods take as training data a set of graded programs and predict students' success or failures on a given assignment. To do that, prior work used syntax-based feature engineering approaches to extract features from the code abstract syntax tree (AST) and feed them into machine learning algorithms. We describe their methods and applications below.

2.a) *Methods.* 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, we explain a recently-applied AST n-Gram feature extraction approach [2, 39] 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].

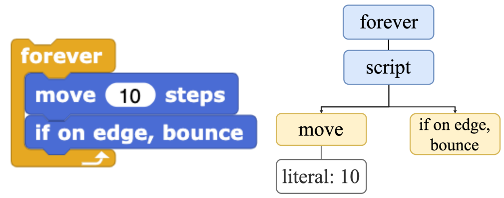


Figure 1: A horizontal n(2)-Gram (in yellow) and a vertical n(2)-Gram (in blue).

Other AST feature engineering approaches were introduced with the usage of Neural Network models, which commonly apply an embedding layer that takes as input multi-dimensional code representations (e.g., a matrix) [3, 13, 22, 32]. In addition, many embedded approaches use sequential models, so that the embedding is trained based on relative locations of code elements [13, 22]. Depending on the Neural Network's architecture, these embedded approaches also vary in ways to represent code as vectors to feed into the models. For example, Alon et al. used leaf-to-leaf feature extraction, connecting the shortest path from each two leaf nodes [5, 4]. They used this approach to represent code and used it for an attention-based model (*code2vec*) to predict method names. The results show that it achieved relatively high accuracy (0.58 F_1 score), and that it learns meaningful vector embeddings that reveal semantic similarities [5].

2.b) *Applications on Student Code Classification.* *1*-Gram and n-Gram-based AST feature extraction approaches have both been applied for student code classification tasks. For example, Azcona et al. applied *1*-Gram feature extractions as a baseline to extract features in 591,707 snapshots of students' Python code, and achieved 0.598 F_1 in predicting students' overall success in completing programming problems, using a simple Naive Bayes model [7]. 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].

Most prior work has used data-driven methods to predict students' success or failures of **the entire problem** [7, 18, 46]. However, to offer students informative feedback, it is important to understand what specific rubric items a student has succeeded or failed in. Some prior work has tried to predict students' success on **specific rubrics**, e.g., by 1) generating rubrics based on a set of correct programs [9, 49]; or by 2) predicting success/failures on rubric items based on training data of a set of graded programs. For example, Shi et al. applied *code2vec* on 207 students' program submissions to predict their success on six different rubric items, in an introductory programming assignment [32]. They found that *code2vec* achieved an average F_1 of 0.69 over all assignments, while a simple SVM model on a *1*-Gram feature extraction achieves an average of 0.46 F_1 score. This work applied data-driven code classification in a real-world, practical setting: in practice, instructors do not have a large amount of training data at hand to train a data-hungry machine learning model (e.g., such as in [7] (591K) and [5] (14M)). [32] shows the potential of applying data-driven code classification not only on a relatively small dataset, but also to predict fine-grained, rubric-specific grades.

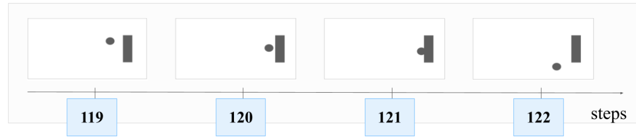
3) **Representing Programs by Execution Traces.** In a systematic literature review on machine-learning-based code analysis, Allamanis et al. critiques that **executability** contributes the key difference between code and natural language, and that **execution traces**, which log variable states during each step of program execution, add an important dimension of analyzing programs by presenting a dynamic viewpoint, and directly linking a piece of programming code to its functionality [3]. In the domain of student code analysis, prior work has identified similar limitations of syntax-based approaches, and pointed to the lack of work of execution-trace-based code analysis [23, 25]. Consequently, recent work that applied execution traces-based code analysis has revealed promising results [23, 45]. For example, 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 [23]. This shows the potential of using execution traces to classify students' programming code.

3. THE ETF APPROACH

The goal of ETF is to extract useful and relevant features from execution traces. It is a feature engineering approach that leverages n-Gram-based feature extraction, which has not previously been applied on execution traces for student code classification. ETF starts from collecting a set of students' programming code, along with a class label for each piece of code, generated from instructor gradings (i.e., positive or negative), and is specifically designed to conduct automated code classification on programs that have the following properties:

1. **Dynamic user interaction.** Programs respond to user interactions (e.g., mouse, keyboard). For example, in games, users use a mouse or keyboard to control actor movements; in an app, users need to navigate to different pages / options using mouse, or keyboard input.
2. **Object-specific program states.** Program states can be represented as properties of objects (e.g., sprites[1]) such as positions and rotations, which correspond to visual output on the screen.
3. **Properties that can change over time.** Program behaviours can be *a function of time* (e.g., movement is described by the actor position that changes over time); in addition, a behaviour can also *change over time* (e.g., move and then stop [34].)

[1]The word "sprite" is used in block-based programming environments, such as Snap*!* and SCRATCH, to represent an object that has its own code (called scripts) and costumes. For example, a sprite can be a game actor or an app widget.



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.

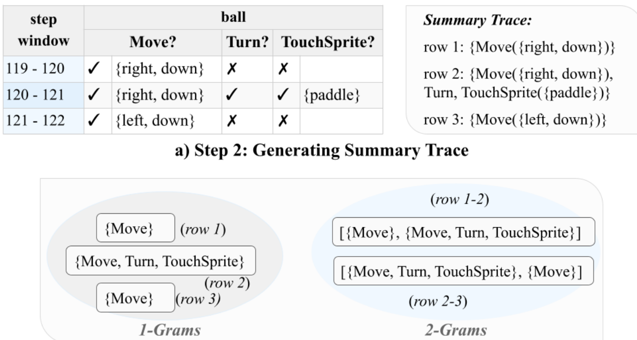
These properties are shared by many different types of visual, interactive program projects, such as games, simulations and apps, and can be easily created in many novice programming environments, such as SCRATCH [29], Snap*!* [21], and Greenfoot [15], as well as programming environments for creative practitioners, such as Processing [28]. ETF conducts feature engineering on these types of visual, interactive programs in four steps, the first three corresponding to the three features above: 1) generating execution traces that provide the inputs for dynamic user interaction (Section 3.1); 2) summarizing traces to include program state-based properties (Section 3.2); 3) extracting features that describe relevant properties that change over time (Section 3.3). The last step 4) is to filter features to generate a $\vec{x}$ vector for each student's code snapshot (Section 3.4).

An Example Assignment. As an example, consider the Pong assignment, which we will use later in our experiments. Pong consists of a paddle sprite and a ball sprite. The ball moves around the stage[2], 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. The Pong assignment is commonly used in a variety of introductory programming courses [47] and camps [17, 31], using various code editors, such as Snap*!* [47], NetsBlox [17], and App Inventor [31], and is therefore representative of many learning contexts and environments. This example assignment includes the three properties of visual, interactive programs that we designed ETF to analyze. We describe the ETF approach below.

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

[2]In Snap*!* and SCRATCH, a stage is a screen on which the program shows its sprite actions.



a) Step 2: Generating Summary Trace

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

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

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`[3]).

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 [35, 41] in the intervals of milliseconds. 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)*** that takes in a sliding window as input, and map this input to a *summarized property set*, which is next logged to the *summary trace*, described below.

The TAF Function. How does TAF map a trace sliding window to a set of properties? 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 is can be an ***abstract property***, describing the changes between steps in the execution trace, such as movement and variable change; 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 position (`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.

- `Move`⟨←, →, ↑, ↓⟩. The `Move` property is returned when a sprite position changed between first and last steps of the sliding window, it uses parameters to describe the direction toward which it is moving.
- `Turn`. The `Turn` property is returned when a sprite changes direction in the sliding window.
- `ChangeSize`⟨`+, -`⟩. The `ChangeSize` property is returned when the Sprite changes its size in the sliding window, it uses parameters to describe whether it changes to bigger (+) or smaller (-).
- `ChangeVar`⟨`variable names`⟨`+, -`⟩⟩. The `ChangeVar` property is returned when one or more global variable's value has been changed between the first and last Step of the sliding window. Its parameters are the actual variables that have changed value. To enable comparison across student programs, the variable names are standardized: the first changed variable in a student's execution trace is named as "var 1", and the next is named as "var 2". Each changed variable also takes in the attribute "+" or"-", to indicate whether the variable is increased or decreased.
- `ChangeLocalVar`⟨`variable names` ⟨`+, -`⟩⟩. The `ChangeLocalVar` property is logged using the same format as the `ChangeVar` property, but only on the sprite-specific variables.

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

[3]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.

power set on the types of properties that are possible in this *2*-Gram. For example, the *2*-Gram generated by rows 2-3 in the property sequence [{`Move`}, {`Move`, `Turn`, `TouchSprite`}] may be divided into 6 subsets as its power sets: [{`Move`}, {`Move`}], [{}, {`Turn`}],[{}, {`TouchSprite`}],[{`Move`}, {`Move`, `Turn`}], [{}, {`Turn`, `TouchSprite`}], [{}, {`Move`, `TouchSprite`}], and the *2*-Gram itself. 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). This creates a representative sample of when students might ask for help—both during programming, and before submitting their code. To create rubric-based labels on all in-progress and submitted code snapshots, one researcher manually graded all 162 programs based on the 10 rubric-based assignment requirements specified by the course instructor: 1) & 2) *key_up / down*: Paddle moves up / down with the up / down arrow key. 3) & 4) *upper / lower_bound*: When touching the upper / lower bound, the paddle does

Table 1: Prevalence of correct programs on each rubric item.

rubric item	prevalence
key_up	0.76
key_down	0.73
upper_bound	0.55
lower_bound	0.56
space_start	0.50
edge_bounce	0.49
paddle_bounce	0.42
paddle_score	0.35
reset_score	0.23
reset_ball	0.31

not move upwards / downwards even when the up / down key is pressed. 5) *space_start*: The ball starts movement when the space key is pressed. 6) *edge_bounce*: The ball bounces when touching the stage edge, unless touching the wall. 7) *paddle_bounce*: The ball bounces when touching the paddle. 8) *paddle_score*: If the ball touches the paddle, points increase by 1. 9) *reset_score*: If the ball touches the wall behind the paddle, the points are reset to 0. 10) *reset_ball*: If the ball touches the wall behind the paddle, the ball is reset.

We selected this dataset for three reasons: 1) Our prior work conducted formative analysis on the 42 submitted snapshots [41], and found it to represent the typical properties that are shared by many visual, interactive programs: dynamic user interaction; object-specific program states; and properties that change over time. 2) It is a **widely-used** introductory programming assignment [47, 17, 31], suitable for many types of programming environments, and is therefore representative of many learning contexts and environments.

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. First, to automatically execute student programs, we started by defining 5 ***input generation rules*** (explained in Section 3.1), corresponding to the requirements of rubric items: 1) *space key once*: ***when*** game starts, press space key once; corresponding to the *space_start* rubric item; 2) & 3) *intermittent up / down key*: press up / down arrow key for 2 steps in every 100 milliseconds, for a total of 15 seconds; corresponding to *key_up* and *key_down* rubric items; 4) & 5) *follow / evade ball*: ***when*** `ball.y` > `paddle.y`, press up key (to follow) / down key (to evade); ***when*** `ball.y` < `paddle.y`, press down key (to follow) / up key (to evade); *follow ball* were defined to ensure the paddle catches the ball, for the *paddle_bounce*, *paddle_score* rubric items; and *evade ball* were defined for the ball to hit back wall, for the *reset_score*, *reset_ball* rubric items. For each program snapshot, we re-executed it 5 times, each time performing the following sequence of inputs: 1,2 (meaning *space key once* followed with *intermittent up key*); 1,3; 1,4; 1,4,5 and 1,4,5. Each run of student programs generated one execution table. Overall, this resulted in 5×162 *execution trace tables*, each with an average of 1182 rows. After generating execution traces, we applied Steps 2 and 3 of ETF Algorithm (Section 3.2 and 3.3) to summarize each execution trace based on 2-step-based sliding windows, creating a *summary trace* with an average of 1181 rows per summary trace. Next, we created *1*-Gram and *2*-Gram-based *full feature sets* for each program. This Step collected from each student an average of 518 distinct features, and created a total of 11148 distinct features in the full feature set.

Generating AST n-Gram and *1*-Gram Features. To examine how well ETF performs, we first compared 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 described by Akram et al. in [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 [27, 32, 48], we used a single label for all literals (`literal`). This Step collected from each student an average of 153 distinct features, and created a total of 1145 distinct features as the full feature set. In addition to the structured n-Gram based approach, we also included a naive baseline, where we only generate the full feature set by extracting *1*-Grams from all students' ASTs (n-Grams with $n = 1$). This is essentially a 1-hot encoding of the AST nodes, where each node is either present (1) or absent (0). Each student's program had an average of 20 distinct *1*-Grams, creating a full feature set with 57 distinct *1*-Grams.

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 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 [36]). 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).

Table 2: 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

4.2.1 Results

Comparison Across Models. For each of the 6 models described earlier, we used the model to predict students' rubric-based performance, and 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. Table 2 reports the average F_1 of each classifier in its prediction across all rubric items. First, an overall 0.73+ F_1 score of the AST *1*-Gram features shows that each node in students' code AST offered relatively useful information to indicate whether the student have successfully completed a certain rubric. Second, we saw that the AST n-Gram approach performed relatively higher than the *1*-Gram approach (5 in 6 cases), but only by 0.002–0.02. In addition, we also saw 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.

Performance Across Rubrics. We next investigate the

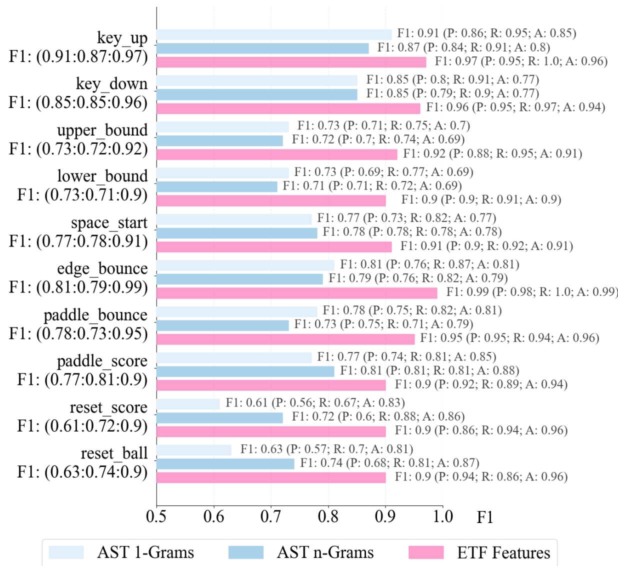


Figure 4: The F_1 score (F1), Precision (P), Recall (R) and Accuracy (A) of ETF, n-Gram, and *1*-Gram features on each rubric, using an SVM model. x-axis starts from $F_1 = 0.5$.

performance 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.

5. DISCUSSION & LIMITATIONS

We found that ETF creates features that lead to highly accurate rubric-based classification of students' in-progress and submitted code (average $F_1 = 0.93$), and it only requires relatively little labeled data to achieve high performance on most rubrics. Therefore, this work shows potential to use the ETF approach to generate formative feedback to students, indicating their progress on each rubric item as they work. This would only require an instructor to manually grade a small amount of students' in-progress and submitted code to create the model. We envision that the ETF approach can be used to build a tool where, whenever a student runs their code, it logs an execution trace, extracts a set of meaningful features from that execution trace, and use the presence and absence of features to accurately predict students' rubric-based performance. This could be used to offer encouraging feedback as students progress that can improve performance and affective outcomes (e.g. [8, 20]). It could also monitor students' progress over time, and prompt them to seek help when they have not progressed for a while (e.g. [19, 30]), or to inform a teacher dashboard summarizing students' progress (e.g. [44]). While our model is not perfect, its accuracy is high enough to ensure the overall quality of this feedback, and comparable with expert-authored formative assessment models, used in existing systems [20, 42], which are also fallible. However, despite the strong potential of applying execution trace-based feature engineering to visual, interactive programs, our work does include several limitations, discussed below:

Limitation: ETF features are sensitive to inputs. All trace-based program analysis takes in inputs to execute the programs. However, defining inputs for visual, interactive programs is particularly difficult. We have shown that defining one set of input generation rules may not always produce high-coverage executions for all programs. This caused ETF to fail to generate features that are indicative of a certain behavior. Most of the erroneous predictions produced by the ETF features were caused by insufficient input sequences, showing that the input sequence had a strong influence on the classification results. In addition, if a student started the project with mouse click button instead of pressing space or green flag, the used input sequence would be insufficient to start students' program, and therefore unable to detect subsequent program behaviors.

This difficulty about producing high-coverage input sequences is shared by all program analysis tools for visual, interactive programs [35, 34, 41, 37], and requires future work such as fully automated input generation to handle this problem. However, we also imagine that the real-world scenario that we envision ETF to work best in does not always require writing *input generation rules* for automated input generation. We may simply log execution traces when a student or instructor manually executes a program (e.g., to test or grade it), and use these execution traces to train the model (from instructor-generated traces) and to offer feedback (for student-generated traces). These execution traces would almost certainly include meaningful inputs that students / instructors use to test and grade programs, which may be even more effective than our input generation rules.

Limitation: Selection of candidate properties require human expertise. For the Pong assignment, we designed ETF to include 9 candidate properties, such as detecting movement and variable changes in the execution trace. While we expect the candidate properties we included to be generalizable, as they are found in many other novice programming tasks, we did not evaluate their generalizability here. In addition, there could potentially be other properties that are important for different assignments (e.g., sound effects, mouse movement, etc), which would require adding new candidate properties. We make no claim that the 9 properties we designed for Pong are an exhaustive list of useful properties, but suggest them as a core set that should be generally useful. In future work, we will explore expanding this set of candidate properties to cover additional assignments and to explore the generalizability of the ETF approach.

6. CONCLUSION

In this work, we explore feature engineering for classifying visual, interactive programs. We present 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.

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] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[4] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A general path-based representation for predicting program properties. *PLDI'18*, 2018.

[5] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *POPL'19*, 2019.

[6] C. R. Association et al. Generation cs: Computer science undergraduate enrollments surge since 2006. *Retrieved March*, 20:2017, 2017.

[7] D. Azcona, P. Arora, I.-H. Hsiao, and A. Smeaton. user2code2vec: Embeddings for profiling students based on distributional representations of source code. In *LAK'19*, 2019.

[8] 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.

[9] N. Diana, M. Eagle, J. Stamper, S. Grover, M. Bienkowski, and S. Basu. Data-driven generation of rubric criteria from an educational programming environment. In *Proceedings of the 8th International Conference on Learning Analytics and Knowledge*, pages 16–20, 2018.

[10] 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.

[11] S. Grover. Toward a framework for formative assessment of conceptual learning in k-12 computer science classrooms. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 31–37, 2021.

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

[13] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[14] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, pages 199–206. IEEE, 2004.

[15] M. Kölling. The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–21, 2010.

[16] 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.

[17] N. Lytle, A. Milliken, V. Cateté, and T. Barnes. Investigating different assignment designs to promote collaboration in block-based environments. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 832âĂŞ838, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Y. Mao, R. Zhi, F. Khoshnevisan, T. W. Price, T. Barnes, and M. Chi. One minute is enough: Early prediction of student success and event-level difficulty during a novice programming task. *International Educational Data Mining Society*, 2019.

[19] S. Marwan, A. Dombe, and T. W. Price. Unproductive help-seeking in programming: What it is and how to address it. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 54–60, 2020.

[20] 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.

[21] J. Moenig and B. Harvey. Byob build your own blocks (a/k/a snap!). *URL: http://byob. berkeley. edu/, accessed Aug*, 2012.

[22] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[23] 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.

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

[25] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *International Conference on Machine Learning*, pages 1093–1102, 2015.

[26] T. Price, R. Zhi, and T. Barnes. Evaluation of a data-driven feedback algorithm for open-ended programming. *International Educational Data Mining Society*, 2017.

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

[28] C. Reas and B. Fry. Processing: programming for the media arts. *AI & SOCIETY*, 20(4):526–538, 2006.

[29] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[30] I. Roll, V. Aleven, B. M. McLaren, E. Ryu, R. S. d Baker, and K. R. Koedinger. The help tutor: does metacognitive feedback improve studentsâĂŹ help-seeking actions, skills and learning? In *International Conference on Intelligent Tutoring Systems*, pages 360–369. Springer, 2006.

[31] K. Roy. App inventor for android: Report from a summer camp. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 283–288, New York, NY, USA, 2012. Association for Computing Machinery.

[32] 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.

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

[34] 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.

[35] 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.

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

[37] W. Wang, G. Fraser, T. Barnes, C. Martens, and T. Price. Automated classification of visual, interactive programs using execution traces. In *Proceedings of the 14th International Conference on Educational Data Mining (EDM 2021)*, 2021.

[38] 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. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 394–400, New York, NY, USA, 2021. Association for Computing Machinery.

[39] 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. *Educational Data Mining in Computer Science Education (CSEDM) Workshop @ EDM'20*, 2020.

[40] 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.

[41] W. Wang, C. Zhang, A. Stahlbauer, G. Fraser, and T. Price. Snapcheck: Automated testing for snap programs. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 227–233, New York, NY, USA, 2021. Association for Computing Machinery.

[42] 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.

[43] 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.

[44] F. Xhakaj, V. Aleven, and B. M. McLaren. Effects of a teacher dashboard for an intelligent tutoring system on teacher knowledge, lesson planning, lessons and student learning. In *European conference on technology enhanced learning*, pages 315–329. Springer, 2017.

[45] L. Yan, N. McKeown, and C. Piech. The pyramidsnapshot challenge: Understanding student process from visual output of programs. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 119–125, 2019.

[46] T. P. T. B. Ye Mao, Samiha Marwan and M. Chi. What time is it? student modeling needs to know. pages 171–182. Proceedings of The 13th International Conference on Educational Data Mining (EDM 2020), 07 2020.

[47] R. Zhi, M. Chi, T. Barnes, and T. W. Price. Evaluating the effectiveness of parsons problems for block-based programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 51–59. ACM, 2019.

[48] 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.

[49] R. Zhi, T. W. Price, N. Lytle, Y. Dong, and T. Barnes. Reducing the state space of programming problems through data-driven feature detection. In *Educational Data Mining in Computer Science Education (CSEDM) Workshop@ EDM*, 2018.

[50] K. Zimmerman and C. R. Rupakheti. An automated framework for recommending program elements to novices (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 283–288. IEEE, 2015.