Early Performance Prediction using Interpretable Patterns in Programming Process Data

Ge Gao
North Carolina State University
Raleigh, USA
ggao5@ncsu.edu

Samiha Marwan
North Carolina State University
Raleigh, USA
samarwan@ncsu.edu

Thomas W. Price
North Carolina State University
Raleigh, USA
twprice@ncsu.edu

ABSTRACT

Instructors have limited time and resources to help struggling students, and these resources should be directed to the students who most need them. To address this, researchers have constructed models that can predict students' final course performance early in a semester. However, many predictive models are limited to static and generic student features (e.g. demographics, GPA), rather than computing-specific evidence that assesses a student's progress in class. Many programming environments now capture complete time-stamped records of students' actions during programming. In this work, we leverage this rich, fine-grained log data to build a model to predict student course outcomes. From the log data, we extract patterns of behaviors that are predictive of students' success using an approach called differential sequence mining. We evaluate our approach on a dataset from 106 students in a block-based, introductory programming course. The patterns extracted from our approach can predict final programming performance with 79% accuracy using only the first programming assignment, outperforming two baseline methods. In addition, we show that the patterns are interpretable and correspond to concrete, effective – and ineffective – novice programming behaviors. We also discuss these patterns and their implications for classroom instruction.

CCS CONCEPTS

• **Social and professional topics → Computer science education**.

KEYWORDS

Student Performance Prediction; Sequential Pattern Mining; Model Interpretation; Student Programming Behavior

ACM Reference Format:
Ge Gao, Samiha Marwan, and Thomas W. Price. 2021. Early Performance Prediction using Interpretable Patterns in Programming Process Data. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432439

1 INTRODUCTION

Many students struggle during introductory programming courses, failing classes [5, 40] or dropping out of CS programs [11]. One-on-one instructor guidance is one of the most effective ways to help struggling students [7], but instructors have limited time to spend with individuals. What if an instructor could know after a student's first programming assignment whether the student will be a low performer in the course? This would allow an instructor to focus their finite resources on helping the students who stand to benefit most from them. It would also allow researchers to personalize automated help (e.g. hints and feedback [36, 37]), to offer higher levels of scaffolding only to students who need it.

Prior work has constructed machine learning approaches to predict students' performance in a given course based on *static* and *general* factors, such as grades in their previous courses, majors, and demographic factors [13, 21]. These models have achieved some predictive success. However, few models use factors specific to Computer Science, such as students' programming behavior. Additionally, by focusing on static features (e.g. GPA, demographics), these models might suggest that a student's potential to succeed is fixed, rather than a dynamic product of their in-class effort. There is also a need for more *interpretable* predictive models, which can tell us not only *whether* a student will succeed, but also offer insight into *why*, with implications for pedagogy.

Many programming environments capture rich log data as students work, including detailed, sequential records of students' interactions with the environment [19, 31, 34]. Prior work has shown that analysis of this *process data* can lead to more accurate predictions of student performance than static measures (e.g. assignment grades) [6, 18, 31]. Log data also captures the same information that an instructor might observe when watching a student program, suggesting potential for data-mined patterns to be interpretable by instructors. Many existing approaches for predicting students' final performance with process data use expert-defined metrics, for example based on students' patterns of compiler errors [4, 9]). However, these metrics require extensive researcher expertise to author manually, and are only weakly predictive of student outcomes [35].

To address these limitations, we propose a novel, *data-driven* approach for predicting student course outcomes based on their *process data* (i.e. interactions with the programming environment). We extract patterns of students' actions from their log data on *early assignments*, which are predictive of their *final* performance in a course. We evaluate our approach in a block-based programming course, finding that some of these patterns correspond to meaningful novice programming behaviors and present case studies of how they manifested in individual students. Specifically, we investigate two research questions:

SIGCSE '21, March 13–20, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432439

RQ1: How do students' patterns of programming predict their final performance in a course, and how early can we make this prediction?

RQ2: How do the patterns generated by such an approach inform our understanding of how students struggle and learn to program?

Our **contributions** are three-fold: (a) we present a sequence mining method for programming log data, based on effective approaches in other domains; (b) we demonstrate that our data-driven approach can generate patterns that predict students final course performance with ~79% accuracy, using log data from just the first assignment; (c) we show how patterns extracted by our approach can inform our understanding of students' programming behavior.

2 RELATED WORK

Many approaches have been proposed for making early predictions of student outcomes in CS courses. These approaches generally consist of two steps: 1) extracting relevant features and 2) training a machine learning model to predict outcomes. Here we focus on the *feature extraction* step. These features can be generally categorized into four types:

Background Features: These are static features about a student or their history before entering the course, such as demographics, GPA, and prior courses taken [13, 21]. For example, ElGamal et al. predict students' programming performance based on their demographics such as gender and prior experience [13]. However, these approaches require that instructors collect this data, which is not always done. Additionally, these features only reflect a student's background, and not their effort in the course, and their predictions may therefore promote an unproductive *fixed mindset*, rather than a *growth mindset* for instructors who use them [29].

Grade Features: These features use students' grades on prior assignments *within a given course* to predict later course outcomes [10, 38]. For example, Castro-Wunsch et al. counted passed and failed assignments as predictors of struggling students [10]. These features do capture students' in-course performance and effort; however, student grades only describe a students' final product, and lose relevant information about *how* the student created it.

Expert-authored features derived from Process Data: These features are *defined by experts* and extracted from students' *process data* as they work, e.g. on programming assignments [2, 10, 14] or clicker questions [25, 33]. They might be simple counts of student actions (e.g. steps taken [2] or submissions [14]), or more complex metrics derived from student action sequences. For example, error metrics, such as the Error Quotient [22], NPSM [8, 9] and others [4, 41] quantify the extent to which students struggle with error metrics based on a sequence of their compilation and other behaviors (e.g. create break points [9]). However, these features require careful expert authoring, based on observations of student behavior, and as such they may not generalize to new datasets, and can have low predictive accuracy [35].

Data-mined features extracted from Process Data: These features are extracted *automatically* (or semi-automatically) from students' programming process data [39], *discovering* behaviors that might be predictive of student success, beyond what experts have identified. For example, Blikstein, Piech, et al. [6, 32] used learning analytics approaches to cluster students' problem-solving trajectories on programming assignments and found these clusters to be predictive of students' final grades, moreso than midterm grades. Mao et al. used temporal pattern mining approaches to extract features from sequences of programming snapshots [27]. These models showed strong predictive success, though they predicted performance on a *single assignment*, rather than across the semester, as in this work. From a theoretical perspective, Grover et al.'s hypothesis-driven framework for learning analytics [17] argues that students' fine-grained interactions can serve as meaningful evidence of specific competencies, arguing that "students' actions need to be detected and measured as students work." They demonstrate how patterns mined from student process data can also be used to inform our understanding of how students express those competencies, though they stop short of using these mined patterns in their assessment models.

In this work, we compare data-mined process data features to a baseline model that uses grade and expert-authored features to determine whether data-driven features improve our ability to predict student outcomes.

3 METHOD

We present χ^2 - differential sequence mining (CDSM), a data-driven feature engineering algorithm, designed to identify sequences of events (i.e. patterns) in students' log data that differentiate two groups of students: high-performing (**HP**) and low-performing (**LP**) students. Specifically, the method identifies two types of patterns: 1) patterns that occur for *more students* in one group than another (e.g. 30% of LP students vs. 60% of HP students), and 2) patterns that occur for students in both groups, but appear *more times* for students in one group than another (e.g. an average 5 times per assignment for HP students vs. 1 time per assignment for LP students). The presence of these patterns can then be used as features in a predictive model.

CDSM builds on the original differential sequence mining (DSM) approach of Kinnebrew et al. [23]. We extend this approach in 3 ways. First, we define an *event categorization* method for extracting discrete, meaningful events from programming process data (Section 3.1). We then extend DSM to detect less frequent, but still discriminating patterns using a χ^2 test (Section 3.2). Last, we use feature discretization to improve the performance of the extracted features in a model (Section 3.3).

3.1 Event Categorization

For each student in class, their log data can be represented as a sequence of discrete events that occurred during programming assignments, along with a label (e.g. high/low performing). The sequence should capture all student interactions with the programming environment (e.g. editing code), as well as other interface actions (e.g. menu clicks). The first challenge for the algorithm is to represent these events with the right level of granularity. For example, if events are too specific (e.g. treating every keystroke as a unique event) it will be difficult to find shared patterns across students, but if they are too general (e.g. one event for all code edits), patterns may not be meaningful.

To address this, we leverage the event categorization scheme of the ProgSnap2 format for programming log data [35] to make our approach more generalizable across datasets, as ProgSnap2 is designed to represent students' interactions within a variety of

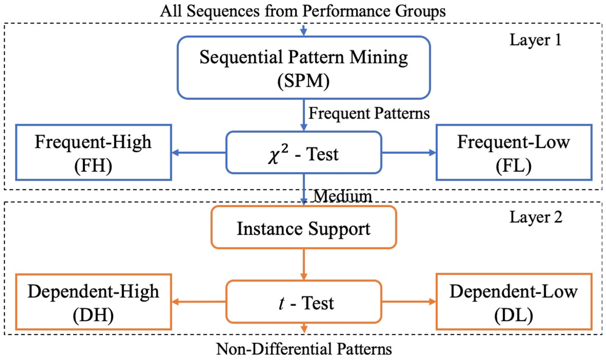


Figure 1: Schematic of the Pattern Mining Algorithm.

programming environments (e.g. block- and text-based) and educational settings. This allows us to build on established event definitions and makes it straightforward to apply our approach to a growing number of programming datasets using ProgSnap2 format. Specifically, we used 5 types of events[1]: 1) **EDIT** (File.Edit): The student edits their program. ProgSnap2 also defines several subcategories of EDIT events, and we treat these as distinct events: EDIT-INS: code is inserted into the workspace, EDIT-DEL: code is deleted from the workspace, and EDIT-PST: code is pasted or duplicated into the workspace; 2) **RUN** (Run.Program): The student runs their program; 3) **FILE** (File.*): The student performs a file-related action, such as closing, saving, and deleting a file. We collapse these various ProgSnap2 events into a single FILE event type; 4) **CHAN** (X-ChangeBlockCategory): The student selects a different block category (events prefixed with 'X-' are specific to our dataset, described in Section 4.1); 5) **VAR** (X-AddVariable): The student creates a new variable.

3.2 Pattern Mining

After categorizing all events in the trace data, all the actions taken by students in each programming assignment can be formulated as a sequence of events. To simplify the representation of sequences, consecutive actions of the same type are collapsed into a single action in this paper (e.g. EDIT RUN is considered instead of EDIT EDIT RUN). This is useful because it collapses similar patterns into one representation, making it easier to find common behaviors among students. However, this also means we may lose some potentially relevant details in the patterns. This choice is designed to strike a balance between specificity and generality. Then the pattern mining step is devised to identify patterns, from these sequences, which can differentiate programming behaviors between two students groups (i.e. HP and LP)regarding their final performance.[2]

To achieve this, we propose a two-layer pattern mining algorithm as depicted in Figure 1. In Layer 1, Sequential Pattern Mining [1] is first applied to find frequent patterns (**FP**s) among all the sequences within each performance group, respectively. Then a χ^2-test is performed to identify two types of patterns – *frequently* high performance (**FH**) or low performance (**FL**) , which contain patterns that *frequently* occurred among HP and LP students respectively. The patterns do not show statistical significance in this layer are assigned to the 'Medium' group and transmitted to Layer 2. Specifically, we extract FPs according to 3 factors – minimum percentile support, maximum gaps and maximum length. Minimum percentile support is the minimum portion of sequences that contains a pattern. Maximum gaps is the maximum number of actions can be skipped when constructing patterns within a sequence. Maximum length refers to the maximum number of actions required to construct a pattern within a sequence. Then, all the patterns whose all three factors meet certain thresholds are selected to be FPs. Next, for each FP, we count the frequency of occurrence (FoC) in the HP and LP groups respectively. χ2-test is employed to investigate whether the FoC of a FP *significantly* correspondeds to any performance groups (i.e. HP or LP). If so, the FP is associated categorized as FH or FL.

In Layer 2, the instance-support [26] of each FP in the Medium group is calculated, followed by a t-test to label it as a dependent-high (**DH**) or dependent-low (**DL**) pattern. Specifically, for each pattern, its instance-supports are calculated across all sequences in HP and LP respectively, which then results in two sets of instance-support (i.e. one set corresponds to HP and the other set corresponds to LP). Then, a t-test is used to determine whether one set is *significantly* different from the other. If so, then it implies that the corresponding FP appears more frequently in one performance group than the other, and it is associated with either DH or DL depending on the mean value of instance-supports within each group (we select the group with higher mean value). If the t-test does not show significance, then the corresponding FP is determined as non-differential patterns and discarded.

3.3 Feature Discretization

As prediction techniques usually require numeric inputs, the patterns we obtained from the pattern mining procedure need to be cast as numerical features. To achieve this, we design a tabular feature representation method. Specifically, we store the FoC of all FPs in the 4 categories in a feature table where each column corresponds to a FP and each row corresponds to a student. We use discretization in this work since it has low computational complexity and has generally shown a good performance on classification tasks [15]. Specifically, for the columns that are associated with FH or FL, all the containing FoCs are discretized into two equal-frequency bins. The cells that are smaller than the median are assigned with 0's, while the others are assigned with 1's. Similarly, for columns belonging to DH and DL groups, all the FoCs are discretized into three equal-frequency bins with values 0, 1, and 2 using 1/3 and 2/3 quantiles. We split them up into 3 bins because the FPs in DH and DL are more common to appear in both student performing groups (i.e. FH and FL) simultaneously.

4 EXPERIMENT

4.1 Population and Data

We use data from an introduction to computing course for non-majors at a large, public university in the Eastern United States. The instructors were not part of the research team. The data was collected across two semesters, Spring 2017 and Fall 2017, with a total of 106 students. In both semesters, students worked on

[1] We used a subset of ProgSnap2 events relevant to our block-based dataset, but additional events, such as "Compile" could be included for other datasets.

[2] Certain criteria (e.g. course final grade) can be used to divide students into these two groups.

the same 5 programming exercises (A1-A5) in a block-based programming environment called Snap. We did not have access to students' demographic information or prior experience, but the course was designed for novices. The programming part of the course was taught in labs led by undergraduate TAs. As students worked, the programming environment automatically logged each action a student took (e.g. adding a code block, running code), including complete snapshots of students' code, allowing researchers to recreate a students' programming process.

Rather than using in-class grades which are made by undergraduate TAs (which vary year-to-year), we regraded the exercises with a similar set of rubrics, using the final snapshot of each student's program, based on the exercise's objectives. We treat the average of students' grades on these exercises as a measure of their programming performance over the whole course, with the grade of each exercise scaled to (0, 1). We used a median split to divide the students into 54 high performers and 52 low performers, based on a median grade of 0.84. As noted in prior work, a median split is appropriate for evaluating the performance of the classification models as well as identifying the patterns that can significantly differentiate high and low performers [3, 16]. This aligns with our goal of identifying students who will perform worse than others and might benefit from prioritized help (not only students 'at-risk' of failing). Our log data is in a well-defined data sharing format, ProgSnap2 [35], and we use it to extract patterns by CDSM. In this study, our data consists of ~530 sequences and ~500 events per sequence.

4.2 RQ1: Performance Prediction

Since RQ1 asks how early we can predict students' grades, we performed 5 separate prediction trials (M1-M5). For a given trial, M_i, we only used data from assignments A_1 through A_i. All trials predicted students' final performance. We extracted patterns from each assignment individually, as completing each assignment requires different programming knowledge, and students' behavior can change as the course progresses. We adopted a *feature stacking* approach, which combines features across assignments to construct a meta feature table. We used an Ada-boost classifier and validated it with a modified hold-out 10-fold cross validation, where 8 sets are used for training, 1 set for testing while the remaining one is discarded.

Our CDSM approach largely uses generic events which exist in any IDE. We wanted to explore how adding IDE-specific context might improve the CDSM approach. We therefore evaluated 2 variants of the Event Categorization approach detailed in Section 3.1: the *general* approach, which only captures generic event information (as described earlier), and the *contextual* approach, which also captures a small amount of Snap-specific contextual information. Specifically, contextual events append a suffix to each action with the name of currently opened block category in Snap (describing what type of blocks a student can add). For example, when an EDIT-INS-pen action is recorded, it means that a student is programming in Snap with *Pen* category opened. Consequently, two explanation levels – *general* and *contextual* – of event categorization techniques can be combined with the CDSM to produce patterns with different levels of explainability, which are denoted as **CDSM-G** and **CDSM-C** respectively.

Table 1: Accuracy, Precision, and Recall of the CDSM-based Prediction Models and Baselines

	Accuracy				Precision				Recall			
	b.1	b.2	G	C	b.1	b.2	G	C	b.1	b.2	G	C
M1	.509	.477	.792	.792	.509	.509	.833	.808	1	.558	.769	.808
M2	.509	.617	.773	.753	.509	.617	.759	.741	1	.698	.830	.811
M3	.509	.677	.850	.774	.509	.684	.852	.778	1	.722	.852	.778
M4	.509	.728	.819	.803	.509	.732	.797	.780	1	.759	.870	.852
M5	.509	.778	.803	.784	.509	.763	.770	.746	1	.833	.870	.870

b.1: Majority; b.2: Expert Rule; G: CDSM-G; X: CDSM-C.

We employ two baselines: 1) *Majority*: using the majority label of performers as predicted label. 2) *Expert Rule*: using students' grades on prior exercises (e.g. grades of A1 and A2 for trial M3), as well as 5 expert-authored process data features: *Block Deletions* (number of blocks deleted), *Block Moves* (number of blocks moved), *Code Runs* (number of program run events), *Time* (the total minutes spent on programming exercises), and *Meaningful Nodes* (the total number of loops, conditionals, and custom blocks made by students in their code). Each of these features has been found to correlate with student performance in previous research (e.g. [9, 14, 20, 24]). Note that Snap does not have compile events, so we could not use compilation error metrics as a baseline. We also employed an Ada-boost classifier with the features from the *Expert Rule* baseline, using the same procedure to tune hyperparameters as our approach.

4.3 RQ2: Pattern Interpretation

We investigated a meaningful subset of the patterns generated by the CDSM approach to evaluate whether they corresponded to meaningful student behaviors. To investigate these patterns, we recorded all patterns extracted from CDSM with their corresponding assignment name and performance group. Then for each pattern, we calculated: 1) portions (i.e. *Perc_High*, *Perc_Low*) of *all* students who have the pattern among high and low performers respectively; 2) the difference (*Diff*) between *Perc_High* and *Perc_Low*; 3) the Odds Ratio (OR) which shows how much more/less likely students who do the pattern are to be low performers. We sorted the patterns by their *Diff* in descending order. Statistically, the higher *Diff* implies the more likely a pattern happens in a specific group. We selected a set of the top 15% of patterns from both performance groups. This helped us identify the most frequent patterns and avoid missing important patterns. For each pattern in the set, 3 researchers replayed the logged actions of individual students who performed the pattern to explore how and in what context the pattern happened during students' programming process. Each researcher summarized their understanding of the pattern independently, then they discussed their interpretations with 2 criteria: 1) what are students doing within the pattern? 2) how might the pattern be indicative of student performance? If all researchers noted that a pattern reflected was indicative of high- or low-performing students, this suggested that the pattern may be meaningful and interpretable.

5 RESULTS AND DISCUSSION

5.1 RQ1: Performance Prediction

The prediction results from Adaboost classification using patterns generated from our approach and 2 baselines are shown in Table 1

Table 2: Pattern Examples and Corresponding Statistics

Index	Ex.	Patterns	PercHigh	PercLow	Diff.	OR
HG1	A2	EDIT-INS CHAN EDIT-INS EDIT CHAN EDIT-INS	52%	31%	21%	2.41
LG1	A1	EDIT-INS EDIT RUN EDIT-INS EDIT RUN	37%	60%	23%	0.34
LG2	A2	CHAN RUN EDIT-INS	22%	42%	20%	0.39

Ex.: the exercise which the pattern is extracted from; PercHigh/Low: the percentage of high/low-performers who have the pattern; Diff.: difference between PercHigh and PercLow; OR: odds ratio.

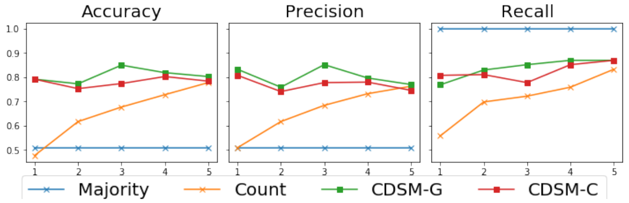


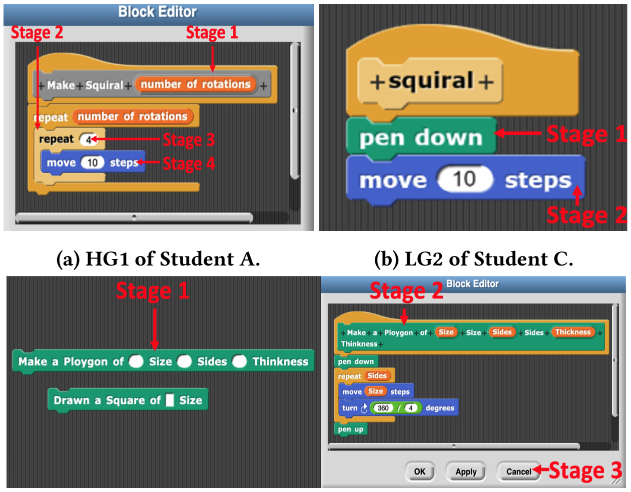
Figure 2: Visualization of Prediction Results by Adaboost. X-Axis: Trial Number; Y-Axis: Prediction Results.

and visualized in Figure 2. Our approach (CDSM-G and -C) performs better with higher accuracy and precision than both baselines among all trials across exercises, except precision on the 5th trial (M5) of CDSM-C. Our approach reaches to 0.792 accuracy within the first exercise. Specifically, using data from only the first exercise, our approach can identify more than 80% of students who will end up as low performers, while miss-identifying less than 20% of high-performers as low-performers (CDSM-G). By contrast, the Count baseline (which included students' actual grades on prior assignments) only reached that accuracy on the last trial, with all available data. We also find that CDSM-C has less accuracy and precision than CDSM-G across exercises, which indicates that contextual information did not improve the prediction.

5.2 RQ2: Pattern Interpretation

From the researchers' examination of patterns, we note that: 1) Not all of the patterns were very informative. For example, the "RUN EDIT-ADD" pattern was more frequent with low performers, but this only indicates that they are more likely to run their code followed by adding blocks, which is a generic behavior to most students. 2) Some of the patterns are highly related (e.g., EDIT-INS CHAN is a subset of EDIT-INS CHAN CHAN) and convey similar information to researchers. We focus on only one pattern from each cluster of similar patterns. Our analysis discovered ∼ 15% distinct, meaningful patterns among the 288 we inspected, from more than 1000 total patterns. Here we focus on case studies of 3 examples (shown in Table 2), where all researchers agreed on their interpretability and meaningfulness.

5.2.1 Case Study 1: Abstraction. The HG1 pattern, EDIT-INS CHAN EDIT-INS EDIT CHAN EDIT-INS, was detected for assignment A2 as more common among high performers. In this pattern, students create a custom block (i.e. a procedure), possibly with parameters, then add blocks from two different categories. The reason why the first EDIT-INS specifically indicates creating a new custom block (procedure), as opposed to adding a new *code block*, is that it is not followed by an EDIT, which always occurs when students "drop" a new code block into their workspace (i.e. stage). HG1 also suggests that students are creating a custom block before they create the code that will go inside of it, as students who move existing code into a newly created custom block would not have the second and third EDIT-INS events. While this pattern may seem overly-specific to one programming environment and its logging approach, CDSM's data-driven approach allows it discover similar niche patterns from other log data too.



(a) HG1 of Student A.

(b) LG2 of Student C.

(c) LG1 of Student B.

Figure 3: Code Construct Examples with Selected Patterns

Student A is a high performer who shows pattern HG1 in A2. Figure 3 shows the code construct that occurred during pattern HG1 for Student A over 10 seconds. It mainly consists of 4 stages of code changes: 1) EDIT-INS: Create a custom block and a variable named "number of rotations"; 2) CHAN EDIT-INS: Change to the *Control* category and add a "repeat" block with default parameter 10 into stage; 3) EDIT: Configure the "repeat" block by updating the parameter to 4; 4) CHAN EDIT-INS: Change to *Motion* category and add a "move" block. In this study, the frequency that this behavior happens among high performers is much higher than low performers (52% vs 31%). This could indicate that students with HG1 understood the importance of custom blocks (i.e procedures), and seemed to have a plan for what to do in that block by adding two new blocks right after.

Students who demonstrated HG1 started by defining a procedure, before adding code, suggesting a more-top down design of their program, or at least more familiarity with how to create procedures. Prior studies related such observations to students' abstraction behavior [42, 43], as procedures allow programmers decompose problems and hide unneeded implementation details.

5.2.2 Case Study 2: Debugging. The LG1 pattern, EDIT-INS EDIT RUN EDIT-INS EDIT RUN, was detected for assignment A1 as more common among low performers than high performers (60% vs 37%). In this pattern, students repeatedly add a new piece of code, then edit their code and run it. Recall that a given event (e.g. EDIT) can represent multiple instances of a type of event occurring in a row, so the events in this sequence may represent a duration of editing or running. Student B is a low performer who has the pattern LG1 in the A1 exercise. Figure 3 shows the code construct captured among student B's changes on their code within 14 seconds when they are doing LG1. It mainly contains 3 stages: 1) EDIT-INS EDIT RUN: Move a block back-and-forth by grabbing one existing custom block "Make a Polygon of # Size # Sides # Thickness [sic]" and snapping it back to the stage. The student then runs code twice but nothing is drawn because of missing parameters inside the procedure; 2) EDIT-INS: Create an incorrect custom block for drawing a polygon because of the wrong usage of "turn" blocks. 3) EDIT RUN: Cancel the custom block and run code again without anything drawn.

We can see that student B runs their program after making one block edit, then they run their program twice after another edit. This is an indicator of debugging behavior with running the program after each single edit. It also suggests that student B does not make progress during the time range when LG1 happens. They move their blocks back-and-forth on stage, which is described as uncertainty or hesitation tinkering behavior by Dong et al. [12], and this is a negative feature of tinkering [30]. When student B gets stuck, they do not appear to stop and spend time to think over about it or seek help from others. Instead, they act unsystematically, which can make the problem worse [30]. Beyond student B's behavior, there are many ways that the pattern LG1 occurs, since it is a general pattern that does not specify block categories. However, it always indicates that a student is running their program frequently within a short period as they enact the pattern. These students may be exhibiting debugging behavior, which has been correlated with lower (B-level) performance [8].

5.2.3 Case Study 3: Testing Blocks Before Adding. The LG2 pattern, CHAN RUN EDIT-INS, was detected in assignment A2 as more common among low performers (42% vs 22%). The pattern consists of a student changing block categories in Snap, then running some code, and then adding new code. For this pattern, it is important to know that in Snap, students can not only run their whole program (by clicking a "green flag"), but run individual blocks by clicking on them. This includes blocks in the "palette", where students can select blocks in opened categories to be added into their code. Manual inspection of the CHAN RUN pattern revealed that the vast majority of the time, it occurs when students switch to a new category and clicked a block in the palette to run it. In other words, the student was not running *their own* code, but running a block they *could add* to their code. This behavior suggests students are unsure what the block does, and may have been searching for a block with a specific functionality. After running the block, the students would often add it to their code. Interestingly, running these blocks sometimes had no visible effect: for example, a "repeat" or "declare variables" block has no output when run alone.

As an example, student C engages in the CHAN RUN pattern twice at the start of their session. The student first changes to the *Pen* category, then clicked on the "pen down" block in the palette which has no visible effect (i.e. they could not see its output). After 6 seconds, they add it to their code. The student then changes to the *Motion* category and runs a "move" block in the palette, which moves the sprite forward. After 3 seconds, they add the "move" block to their code as well. While these "pen down" and "move" blocks are relatively common, it is important to remember that the student has only used these blocks in one prior exercise (A1), so they are still relatively new. This behavior was almost twice as common among low performers on A2, suggesting a hypothesis that testing a block before adding it to their code is an evidence of uncertainty or struggle. In addition to the most common case, described above, LG2 also occurs when students run their custom block by clicking on it in the block palette, rather than by adding it to a script in their own code to run it.

Similar observations have been made in prior work. For example, Zhi et al. [44] suggest that Parsons problems – where students are given all the elements of a correct code solution and must rearrange them – may be effective in part because students do not waste time identifying the code elements they need to solve a problem. This may be particularly true in block-based programming environments, where students must search for new blocks across a large palette. Our results support this interpretation, and they also suggest that uncertainty about the blocks' functionalities, evidenced by the CHAN RUN pattern, may exacerbate this challenge for novices. Mitrovic et al. [28] suggest that programming environments can address this uncertainty with positive feedback, e.g. informing students that they have selected the correct block and made progress.

6 CONCLUSION AND LIMITATIONS

Overall, the **contributions** of this work are: (a) we adapt a sequence mining method to programming log data; (b) we demonstrate that our data-driven approach can generate patterns that predict students final course performance with accuracy ~79%, using log data from just the first assignment; (c) we show how patterns extracted by our approach can inform our understanding of students' programming behavior.

There are two primary limitations in this study: 1) The specific patterns we find are not all generalizable, with some specific to Snap. However, while the individual patterns may not generalize to all contexts, the method should generalize to any context where we have event-level data with ProgSnap2 format. The patterns and our model are evaluated by hold-out 10-fold cross validation, which suggests we will achieve similar results on a new dataset from a similar population. Additionally, many of these events are general across both block-based and textual programming environments. 2) The data does not come from the typical CS class (only 5 programming assignments), and we do not use any external measures (e.g. quizzes). Future work is needed to evaluate the approach in more classroom contexts, including more traditional CS1 courses.

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*. IEEE, 3–14.

[2] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. 2015. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*. 121–130.

[3] Carole Ames and Jennifer Archer. 1988. Achievement goals in the classroom: Students' learning strategies and motivation processes. *Journal of educational psychology* 80, 3 (1988), 260.

[4] Brett A Becker. 2016. A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 296–301.

[5] Jens Bennedsen and Michael E Caspersen. 2019. Failure rates in introductory programming: 12 years later. *ACM Inroads* 10, 2 (2019), 30–36.

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

[7] Benjamin S Bloom. 1984. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher* 13, 6 (1984), 4–16.

[8] Adam Scott Carter and Christopher David Hundhausen. 2017. Using programming process data to detect differences in students' patterns of programming. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 105–110.

[9] Jason Carter, Prasun Dewan, and Mauro Pichiliani. 2015. Towards incremental separation of surmountable and insurmountable programming difficulties. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 241–246.

[10] Karo Castro-Wunsch, Alireza Ahadi, and Andrew Petersen. 2017. Evaluating neural networks as a method for identifying students in need of assistance. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*. 111–116.

[11] Xianglei Chen. 2013. STEM Attrition: College Students' Paths into and out of STEM Fields. Statistical Analysis Report. NCES 2014-001. *National Center for Education Statistics* (2013).

[12] Yihuan Dong, Samiha Marwan, Veronica Catete, Thomas Price, and Tiffany Barnes. 2019. Defining tinkering behavior in open-ended block-based programming assignments. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 1204–1210.

[13] AF ElGamal. 2013. An educational data mining model for predicting student performance in programming course. *International Journal of Computer Applications* 70, 17 (2013), 22–28.

[14] Andrew Emerson, Fernando J Rodríguez, Bradford Mott, Andy Smith, Wookhee Min, Kristy Elizabeth Boyer, Cody Smith, Eric Wiebe, and James Lester. 2019. Predicting early and often: Predictive student modeling for block-based programming environments. In *Proceedings of The 12th International Conference on Educational Data Mining (EDM 2019)*, Vol. 39. ERIC, 48.

[15] Eibe Frank and Ian H Witten. 1999. Making better use of global discretization. In *16th International Conference on Machine Learning (ICML 99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 115–123.

[16] Kaushik VSN Ghantasala, Raeed H Chowdhury, Uday Guntupalli, Jason R Hagerty, Randy H Moss, Ryan K Rader, and William V Stoecker. 2013. The Median Split Algorithm for Detection of Critical Melanoma Color Features.. In *VISAPP (1)*. 492–495.

[17] Shuchi Grover, Satabdi Basu, Marie Bienkowski, Michael Eagle, Nicholas Diana, and John Stamper. 2017. A framework for using hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming environments. *ACM Transactions on Computing Education (TOCE)* 17, 3 (2017), 1–25.

[18] Allyson F Hadwin, John C Nesbit, Dianne Jamieson-Noel, Jillianne Code, and Philip H Winne. 2007. Examining trace data to explore self-regulated learning. *Metacognition and Learning* 2, 2-3 (2007), 107–124.

[19] Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Exploring problem solving paths in a Java programming course. (2014).

[20] Pei-Lun Hsu, Robert Lai, and CC Chiu. 2003. The hybrid of association rule algorithms and genetic algorithms for tree induction: an example of predicting the student course performance. *Expert Systems with Applications* 25, 1 (2003), 51–62.

[21] Qian Hu and Huzefa Rangwala. 2019. Reliable deep grade prediction with uncertainty estimation. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*. 76–85.

[22] Matthew C Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*. 73–84.

[23] John S Kinnebrew, Kirk M Loretz, and Gautam Biswas. 2013. A contextualized, differential sequence mining method to derive students' learning behavior patterns. *JEDM| Journal of Educational Data Mining* 5, 1 (2013), 190–219.

[24] Juho Leinonen, Leo Leppänen, Petri Ihantola, and Arto Hellas. 2017. Comparison of time metrics in programming. In *Proceedings of the 2017 acm conference on international computing education research*. 200–208.

[25] Soohyun Nam Liao, Daniel Zingaro, Kevin Thai, Christine Alvarado, William G Griswold, and Leo Porter. 2019. A robust machine learning technique to predict low-performing students. *ACM Transactions on Computing Education (TOCE)* 19, 3 (2019), 1–19.

[26] David Lo, Siau-Cheng Khoo, and Chao Liu. 2008. Efficient mining of recurrent rules from a sequence database. In *International Conference on Database Systems for Advanced Applications*. Springer, 67–83.

[27] Ye Mao, Samiha Marwan, Thomas W Price, Tiffany Barnes, and Min Chi. 2020. What Time is It? Student Modeling Needs to Know. In *In proceedings of the 13th International Conference on Educational Data Mining*.

[28] Antonija Mitrovic, Stellan Ohlsson, and Devon K Barrow. 2013. The effect of positive feedback in a constraint-based intelligent tutoring system. *Computers & Education* 60, 1 (2013), 264–272.

[29] Laurie Murphy and Lynda Thomas. 2008. Dangers of a fixed mindset: implications of self-theories research for computer science education. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*. 271–275.

[30] David N Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of learning in novice programmers. *Journal of Educational Computing Research* 2, 1 (1986), 37–55.

[31] Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. 2015. Deep knowledge tracing. In *Advances in neural information processing systems*. 505–513.

[32] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 153–160.

[33] Leo Porter, Daniel Zingaro, and Raymond Lister. 2014. Predicting student success using fine grain clicker data. In *Proceedings of the tenth annual conference on International computing education research*. 51–58.

[34] Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 483–488.

[35] Thomas W Price, David Hovemeyer, Kelly Rivers, Ge Gao, Austin Cory Bart, Ayaan M Kazerouni, Brett A Becker, Andrew Petersen, Luke Gusukuma, Stephen H Edwards, et al. 2020. Progsnap2: A flexible format for programming process data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 356–362.

[36] Thomas W Price, Rui Zhi, and Tiffany Barnes. 2017. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *International Conference on Artificial Intelligence in Education*. Springer, 311–322.

[37] Kelly Rivers and Kenneth R Koedinger. 2013. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, Vol. 50.

[38] Cristóbal Romero, Sebastián Ventura, Pedro G Espejo, and César Hervás. 2008. Data mining algorithms to classify students. In *Educational data mining 2008*.

[39] Wengran Wang, Yudong Rao, Yang Shi, Alexandra Milliken, Chris Martens, Tiffany Barnes, and Thomas W Price. [n.d.]. Comparing Feature Engineering Approaches to Predict Complex Programming Behaviors. ([n. d.]).

[40] Christopher Watson and Frederick WB Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 39–44.

[41] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2013. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th international conference on advanced learning technologies*. IEEE, 319–323.

[42] Jeannette M Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (2006), 33–35.

[43] Nong Ye and Gavriel Salvendy. 1996. Expert-novice knowledge of computer programming at different levels of abstraction. *Ergonomics* 39, 3 (1996), 461–481.

[44] Rui Zhi, Min Chi, Tiffany Barnes, and Thomas W Price. 2019. Evaluating the Effectiveness of Parsons Problems for Block-based Programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 51–59.