Progression Trajectory-Based Student Modeling for Novice Block-Based Programming

Fahmid Morshed Fahid
North Carolina State University
ffahid@ncsu.edu

Xiaoyi Tian
University of Florida
tianx@ufl.edu

Andrew Emerson
North Carolina State University
ajemerso@ncsu.edu

Joseph B. Wiggins
University of Florida
jbwiggi3@ufl.edu

Dolly Bounajim
North Carolina State University
dbbounaj@ncsu.edu

Andy Smith
North Carolina State University
pmsmith4@ncsu.edu

Eric Wiebe
North Carolina State University
wiebe@ncsu.edu

Bradford Mott
North Carolina State University
bwmott@ncsu.edu

Kristy Elizabeth Boyer
University of Florida
keboyer@ufl.edu

James Lester
North Carolina State University
lester@ncsu.edu

ABSTRACT

Block-based programming environments are widely used in computer science education. However, these environments pose significant challenges for student modeling. Given a series of problem-solving actions taken by students in block-based programming environments, student models need to accurately infer problem-solving students' programming abilities in real time to enable adaptive feedback and hints that are tailored to students' abilities. While student models for block-based programming offer the potential to support student-adaptivity, creating student models for these environments is challenging because students can develop a broad range of solutions to a given programming activity. To address these challenges, we introduce a progression trajectory-based student modeling framework for modeling novice student block-based programming across multiple learning activities. Student trajectories utilize a time series representation that employs code analysis to incrementally compare student programs to expert solutions as students undertake block-based programming activities. This paper reports on a study in which progression trajectories were collected from more than 100 undergraduate students engaging in a series of block-based programming activities in an introductory computer science course. Using progression trajectory-based student modeling, we identified three distinct trajectory classes: Early Quitting, High Persistence, and Efficient Completion. Analysis of these trajectories revealed that they exhibit significantly different characteristics with respect to students' actions and can be used to accurately predict students' programming behaviors on future programming activities compared to competing baseline models. The findings suggest that progression trajectory-based student models can accurately model students' block-based programming problem solving and hold potential for informing adaptive support in block-based programming environments.

CCS CONCEPTS

• **Social and professional topics** → Professional topics; Computing education; • **Applied computing** → Education.

KEYWORDS

Trajectory-based Student Modeling, Block-based Programming, Time Series Clustering

ACM Reference Format:
Fahmid Morshed Fahid, Xiaoyi Tian, Andrew Emerson, Joseph B. Wiggins, Dolly Bounajim, Andy Smith, Eric Wiebe, Bradford Mott, Kristy Elizabeth Boyer, and James Lester. 2021. Progression Trajectory-Based Student Modeling for Novice Block-Based Programming. In *Proceedings of the 29th ACM Conference on User Modeling, Adaptation and Personalization (UMAP '21), June 21–25, 2021, Utrecht, Netherlands.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3450613.3456833

1 INTRODUCTION

Introductory computer science courses can be challenging for novice students who have limited or no prior programming experience, resulting in high failure and attrition rates [2]. An important ability that students must develop is computational thinking, a complex mental process that involves logically formulating and solving problems [58]. In addition to computational thinking, traditional text-based programming courses also teach students about language-specific syntax and structure. Novice students typically struggle with programming language syntax and structure, which results in a steep learning curve in their introductory courses [55]. To ease the learning process, block-based programming languages such as Scratch [46], Snap! [14] and Blockly [13] are frequently

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UMAP '21, June 21–25, 2021, Utrecht, Netherlands
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8366-0/21/06…$15.00
https://doi.org/10.1145/3450613.3456833

used in introductory computer science courses [29], eliminating many of the common syntactic, language-specific challenges. These visual, block-based languages can reduce students' cognitive load while increasing their engagement in the learning process [41, 56]. However, even with the use of block-based programming languages, students still struggle in introductory classes [30, 42].

Informing student-adaptive support in the form of tailored advice, guidance, hints, and feedback for block-based programming calls for the creation of student models that can accurately analyze student problem solving in block-based programming environments. Programming tasks are typically iterative in nature where programmers progress through the phase of planning, writing, testing, and revising their code. To better understand this iterative process, researchers have studied programming trajectories, which represent the incremental code changes in a programming activity. To understand such programming trajectories, several code analysis methods have been used [9, 34, 53], to provide better support to students [7, 42, 48, 57], develop automatic grading [8, 54], or predict learning performance [11, 53, 54]. Some studies also found patterns in students' programming trajectories that can be used to identify learning behaviors [12, 22, 39]. However, limited work has investigated students' programming trajectory patterns across multiple programming activities to enable early prediction of their learning behavior in future programming activities.

In this work, we introduce *progression trajectory-based student models* that utilize a time series representation employing code analysis to incrementally analyze student programs as students create them. We explore progression trajectory-based student models for the programming trajectories of 149 undergraduate students in an introductory computer science course using a block-based programming environment across five programming activities. We investigate the following three research questions (RQs):

- RQ1: Are there distinct patterns in student's block-based progression trajectories?
- RQ2: Are these patterns associated with meaningful differences in programming behaviors and learning outcomes?
- RQ3: Are these patterns predictive of students' future programming behavior (i.e., progression trajectory)?

To answer RQ1, we analyzed a total of 745 progression trajectories across five different programming activities using time series-based clustering. The analyses revealed that there are three distinct clusters in students' progression trajectories. To understand how these clusters differ from one another (RQ2), we used different response variables across the clusters and found that the clusters exhibit significantly different characteristics across programming activity completion, hint requests, and activity restarts. We also found that students primarily associated with a particular cluster across multiple programming activities have significantly different pre- and post-test scores than students associated with other clusters. To answer RQ3, we trained several predictive classifiers using features collected from student programming trajectories and event logs. The prediction results showed that the models can accurately predict whether a student will change their behavior (i.e., transition to a different cluster) in subsequent programming activities.

2 RELATED WORK

Recent years have seen increasing adoption of block-based programming environments in introductory computer science courses. However, limited work has investigated student models that infer student programming competences from student behaviors in block-based programming environments. Price et al. created iSnap [43], a block-based programming environment that can generate adaptive contextual hints using students' programming behavior. Further studies showed that such hints can lead to positive learning outcome in undergraduate-level introductory programming courses [42] and to better help-seeking behavior as well as reduction of hint abuse [44]. Specific hint types have also been studied. For example, Zhi et al., found that worked examples reduce cognitive load but do not result in significant learning gains [59]. Several other studies with students in MOOCs showed the efficacy of data-driven approaches to provide effective feedback to students [19, 28, 31, 36]. Despite these advances, a recent study comparing machine-generated feedback to human-authored feedback showed that students develop a better conceptual understanding of programming concepts and perform better in class when assisted by human-authored feedback [25].

Student modeling can provide insight into students' future learning behaviors and performance. For example, Emerson et al. showed that Bayesian hierarchical student models can account for individual student differences and successfully predict students' post-test programming performance [9]. Other work has used Bayesian student models to identify concepts that are difficult for students [47] and to predict mastery learning [38, 53]. Machine learning approaches were found to be effective in modeling students' competencies and goals in game-based learning environments using game trace data [23, 32, 33]. Multimodal data streams have also been used to model students' performance and affective states [18, 27]. Barria-Pineda et al. designed open learner-based student modeling to visually inform students about their conceptual knowledge level on topics and found that students make more informed decisions when navigating through the programming activities while increasing their overall learning performance [1]. Another study by Troiano et al. clustered the developments of programming concepts among K-12 students in Scratch using automated metric-based assessments and found four groups of progressions: quick-then-steady, steady-all-the-way, slow-and-still-developing, no-improvement-necessary [52]. In our work, we have modeled students' progress towards solutions using their block-based programs to find and predict their progress patterns in future activities.

Modeling students in open-ended programming activities is critical as there are often multiple paths to a solution, and program accuracy alone does not reveal students' learning behavior [17]. An empirical study by Hosseini et al. showed that intermediate programming steps can be used to effectively induce student models and can be useful for providing improved feedback to students [20]. The authors analyzed the interaction logs of JavaParser and found that there is a strong association between conceptual change (i.e., start with a small program and gradually add new programming concepts) and the correctness of the program. A similar association between programming concepts and performance has been found by using programming trajectories [4, 40]. In another empirical

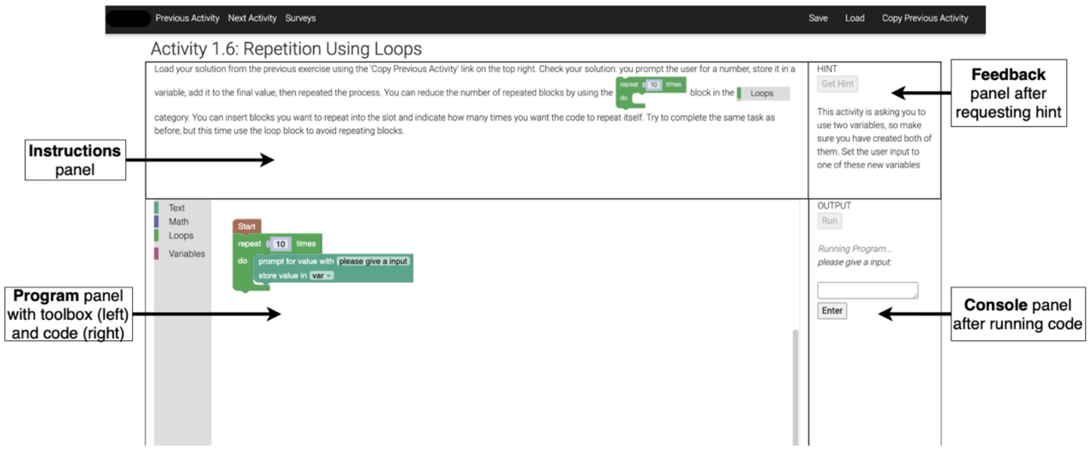

Figure 1: Screenshot of the Prime environment.

study, Blikstein found three patterns of programming behaviors among students, namely copy-paste, self-sufficient, and a combination of both [3]. A recent study by Estey et al. found that intermediate programming steps can successfully identify struggling students [12]. To understand how a student changes their programming pattern in a programming activity, Glassman et al. used a clustering technique to visualize students' behaviors [15].

Much prior work has used edit distance-based code embeddings to investigate students' programming behaviors. For example, Paaßen et al. generated adaptive hints in student programs using their edit distance to expert solutions to predict future behaviors of students in block-based programming activities [35]. A similar approach has been used in other work [48] to investigate programming patterns. Emerson et al. clustered student programs to identify common misconceptions among students in block-based programming activities [10]. Another clustering study by Perkins et al. identified three kinds of novice programmers: *stoppers*, *movers*, and *tinkerers*. As the cluster names suggest, *stoppers* give little effort after they reach a difficult point, whereas *movers* gradually work towards a solution, and *tinkerers* iterate frequently with the program [39]. Similar types of clusters have also been identified in an analysis of program syntax errors [21] and learning behaviors in an intelligent logic tutor [30]. Jiang et al. conducted a hierarchical clustering of programming trajectories in a block-based programming environment and found four different clusters for a single programming activity, namely: *quitters*, *approachers*, *knowers*, and *solvers* [22]. Their analysis suggests that both *quitters* and *approachers* struggled to solve the given programming activity but have different paths while failing. A more recent study found five different clusters of hint usage when looking across several programming activities of students in a block-based programming environment [57], all of which have different help-seeking behaviors. For example, students with prior programming knowledge only requested hints at the very end of each programming activity, whereas students with no prior knowledge asked for hints uniformly. However, there has been limited work on student modeling for students' programming trajectories across multiple programming activities to predict their progress.

In this work, we present a student modeling approach based on students' progression trajectories using intermediate block-based programs on multiple programming activities by iteratively calculating edit distances between their intermediate code states and expert solutions. Using time series clustering, we find meaningful patterns in student progression trajectories that are predictive of future programming behavior.

3 METHODS

We investigate progression trajectory-based student modeling in the context of introductory college-level computer science with a focus on block-based programming for non-majors. We use interaction data collected from undergraduates interacting with a block-based programming environment during the first few weeks of an introductory programming course.

3.1 Prime Learning Environment

Prime is an adaptive learning environment that leverages Google's block-based programming framework, Blockly [13]. The adaptive learning environment includes twenty programming activities, each designed to reinforce the skills learned from previous programming activities. Students are encouraged to complete activities in increasing order, but they can skip ahead and come back later to attempt or practice again. The programming activities are organized into three units with the following computer science competencies: 1) input/output, numeric data types, expressions, variables, and iteration; 2) abstraction, functions, and parameters; and 3) Boolean data types, conditionals, indefinite iteration, and debugging. Students are encouraged to use the learning environment both during class, as well as for self-practice, lab, and homework assignments. The curriculum of Prime was designed based on the introductory programming courses taught in top-rated undergraduate computer science programs in the United States [51].

Table 1: Short descriptions of each programming activity

Programming Activity	New Concepts Introduced	Short Description
3	Variable, Math Expression	Print the sum of two numbers using variables
4	Input	Take two numbers from user and print the sum
5	Logical	Take five numbers from user using two variables only and print the sum
6	Loop	Complete Programming Activity 5 using a for loop
7	Logical	Create a counter that countdown to 0

To support students during their programming activities, the primary interface of the environment consists of a Program panel, a Console panel, a Feedback panel, and an Instruction panel (Figure 1). The Program panel consists of a toolbox of Blockly program blocks that can be drag-and-dropped to the coding workspace. The toolbox categorizes the blocks by type. The categories of blocks are incrementally populated to reduce the cognitive complexity of tasks [45] and to increase the interface usability [49].

The default workspace contains an unremovable *Start* block that serves as an entry point. The *Console* panel contains a *Run* button to run the program connected to the *Start* block. The output is printed on the *Console* panel each time the *Run* button is pressed. The input console is also shown in the *Console* panel when needed. The *Instruction* panel contains step-by-step instructions for each programming activity and is fairly common in intelligent tutoring systems for computer science education [6]. The *Feedback* panel provides a "Get Hint" button that enables students to request textual hints, which are suggestions to make small changes to the program towards a solution.

3.2 Study Design

We collected student programming interaction data on block-based programming activities at a large public university in the United States during two semesters (fall 2019 and spring 2020). The study was IRB-approved for an introductory engineering course, in which students received extra credit for completing the programming activities. As the learning environment was accessible online, transitioning to remote learning was straightforward when the pandemic began in spring 2020. The study focused on five programming activities from Unit 1 (Programming Activities 3 through 7) that teaches students the concept of variables, input/output and loops (Table 1). We focused on these programming activities because they are sufficiently challenging (62% average completion rate) while ensuring maximum participants. We did not use the first two activities (Programming Activities 1 and 2), as they serve as an introduction to the programming environment and have a very high completion rate (99% and 87% respectively). Although Programming Activity 3 also has a significantly high competition rate (83%), we included this programming activity in the study to facilitate early prediction of Programming Activity 4.

A total of 407 students participated in the study, of which 156 students attempted all five programming activities in Unit 1 (Programming Activities 3 through 7). Among these 156 students, 149 students also completed both pre-test and post-test surveys to assess their computer science competencies. The study focuses on these 149 participants having an average age of 19.1 (*SD*=2.2), with 74% reporting as male and 26% as female. Approximately 90% of the students were non-computer science engineering majors. A total of 62% of these students reported their race as White, 16% as Asian, 9% as African American, 6% as Hispanic, 2% as Native American, and 4% as Other. The 149 students attempted a total of 745 programming activities (one student per activity for Programming Activities 3 through 7) of which 459 (62%) were successfully completed. Among these five programming activities, the median value of successful programming activity completion per student is 3 (*M*=3.08, *SD*=1.57, *min*=0, *max*=5).

3.3 Dataset

The data was collected from student interaction logs with the learning environment. The data consists of workspace snapshots captured with an XML representation of student programs of each new change made in the code and 16 system-level user actions, including program runs and user input. The workspace snapshots represent the current block-based program for a particular programming activity. The environment automatically takes workspace snapshots every 30 seconds or if certain actions are taken, such as running or saving the program. We removed repeated snapshots of the same workspace to limit redundancy. As this is a browser-based environment, the calculation of elapsed time is challenging because students sometimes leave their browser open for extended periods of time without making any changes to the workspace. To correct for such cases, we replaced any time gaps of more than five minutes between two consecutive workspace snapshots with the median value (29.96 seconds) of consecutive time gaps between snapshots. The five-minute threshold was selected based on the third quartile value of time gap between two consecutive workspace snapshots. The final dataset contains a total of 745 trajectories, one for each student-activity pair, having a median length of 10 snapshots per programming activity (*M*=19.62, *SD*=29.95). The time spent on each programming activity on average is 508.98 seconds (*SD*=850.38) with a median value of 216.05 seconds.

The dataset also uses students' pre-test and post-test survey before and after interacting with the learning environment to measure their computer science competencies. These multiple-choice based assessments were validated by three content area experts, having 0.88 Cronbach's alpha for pre-test and 0.90 Cronbach's alpha for post-test. Among these 149 students, the median value of the pre-test score was 58.33 (*M*=57.5, *SD*=22.6, *min*=12.5, *max*=100) and the median value of their post-test score was 66.7 (*M*=61.5, *SD*=23.9, *min*=4.2, *max*=100).

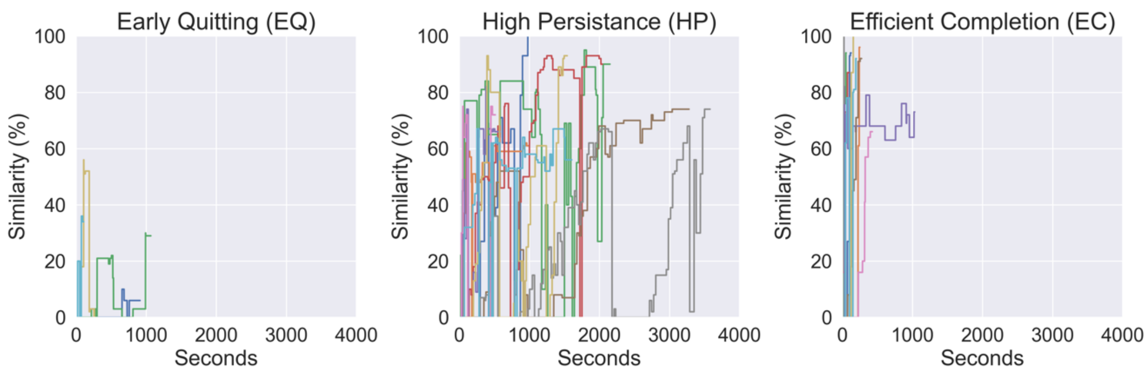

Figure 2: Ten randomly sampled progression trajectories for each of the three clusters. The x-axis is the time in seconds and the y-axis is the similarity percentage between an expert solution and the current workspace (higher is better).

4 RESULTS

We describe findings on the time series-based progression trajectory clustering and early predictions. We first introduce the progression trajectory-based student modeling and then discusses the time series-based clusters we have found. We then investigate and characterize each of the clusters based on multiple attributes. Finally, we evaluate two early prediction-based modeling tasks that can be used to predict students' future programming behaviors.

4.1 Progression Trajectory Patterns

To generate the progression trajectory of a programming activity attempt, we calculated the similarity percentage between workspace snapshots and an expert authored solution for that activity. Each workspace snapshot represents one block-based programming code state whereas the expert-authored solution is the final workspace snapshot of the solution to a particular programming activity. The expert-authored solutions were based on the block-based programming best practices such as minimizing the number of steps towards each solution. To calculate the similarity percentage, we first converted the student's workspace snapshots and the expert solution into Python code. Then, using the program's abstract syntax tree (AST), we calculated the edit distance using Damerau–Levenshtein [5] distance between the workspace and the expert solution. The Damerau–Levenshtein distance is calculated by the minimum number of operations (*insert*, *delete*, *substitute*, *transpose*) needed to convert one string to another string. Finally, the edit distance was converted to a percentage to calculate the similarity between the current workspace and the expert solution. For converting Python codes to ASTs and calculating the similarity percentages, we used the pyastsim package.[1] We used the edit distance to calculate similarity because the majority of the programming activities and their corresponding Python code are relatively simple. Similar approaches have been effectively utilized in prior studies to observe students' progression through individual programming activities [22].

As there can be many correct solutions to most programming activities, some students who correctly completed a programming activity may not have 100% similarity with the expert solution. To ensure these snapshots were treated as correct, we updated their final similarity score to 100% for all the programming activities that were completed successfully. We also removed any change made in their workspace after reaching 100% similarity to focus on their first correct solution.

We clustered the progression trajectories using k-means clustering. As the progression trajectories are time series data with varying lengths and shapes, using generic distance measures will not reflect the true distance between two trajectories. Thus, we used the dynamic time warping (DTW) algorithm to calculate the distance between trajectories. DTW is a dynamic programming approach where a distance or cost function (typically Euclidean distance) is minimized by finding an optimal alignment. This can be done by rearranging and/or repeating sequence points of two sequential trajectories [50].

The distortion elbow is often used as a data-driven approach for finding the optimal number of clusters in a dataset [24]. Thus, to find the optimal value of k, we calculate the distortion elbow using the average DTW distance from the closest cluster center. Visually inspecting the "elbow" (the inflection point of the curve), we chose 3 as the optimal number of clusters of progression trajectories. We name these clusters *Early Quitting (EQ), High Persistence (HP), and Efficient Completion (EC)*. Ten randomly sampled progression trajectories from each cluster are shown in Figure 2. Note that each line represents one student-activity pair and one student's progression trajectories for different programming activities could be assigned to different clusters. With regard to the RQ1, we found three distinct time-series clusters: *Early Quitting (EQ), High Persistence (HP), and Efficient Completion (EC)* in students' progression trajectories.

As show in Figure 2, all three clusters have distinct progression trajectories. For example, progression trajectories in the *HP cluster* spend a large amount of time (*M*=1756.69 seconds, *SD*=1514.82) and change their similarity percentage frequently, whereas progression trajectories in the *EQ cluster* do not make as much progress within their shorter period of time in the programming activity (*M*=195.56 seconds, *SD*=358.59), and progression trajectories in the *EC cluster* quickly make progress towards the solution like an

[1] https://pypi.org/project/pyastsim/

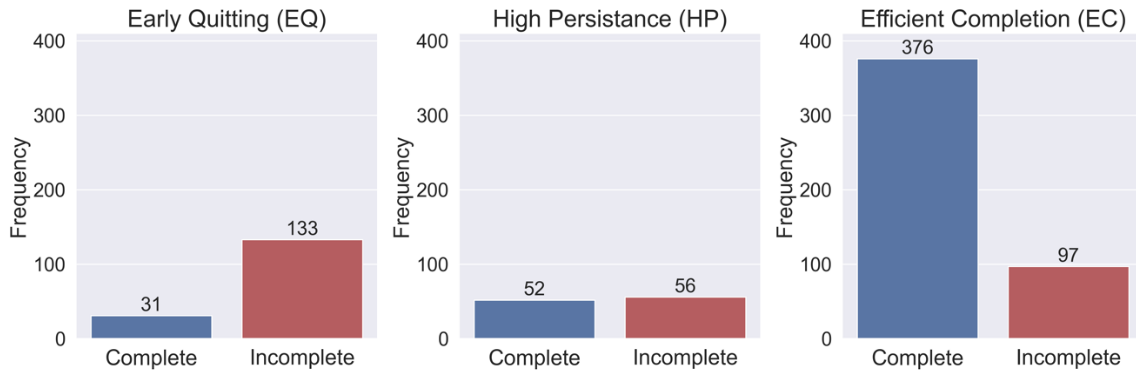

Figure 3: Number of progression trajectories that completed and did not complete each programming activity for cluster EQ, HP, and EC (from left to right)

expert (M=319.53 seconds, SD=371.10). The *EC cluster* has almost twice as many student-activity pairs (473) than the *HP cluster* and the *EQ cluster* combined (108 and 164 respectively). This is reasonable, as the programming activities (e.g., Programming Activities 3 and 4) are introductory and short, and many students were able to make progress through them quickly (as seen by the *EC cluster*).

4.2 Analyzing Clusters

The three clusters that separate students' progression trajectories into different groups have distinct properties. For example, the *EQ cluster* could be characterized as lack of persistence/quick-to-dropout, which might indicate the student is stuck or unsure how to begin solving the programming activity, while the *EC cluster* might indicate high prior knowledge. A deeper analysis of the impact of different characteristics on each individual cluster is detailed below.

First, we investigated the overall performance of each cluster based on successfully completing each programming activity. Figure 3 shows that the *EQ cluster* has a higher number of programming activities with incomplete count; thus, a progression trajectory that belongs to the *EQ cluster* is most likely to leave the programming activity unfinished. The *HP cluster* has an almost equal number of completed vs. incomplete programming activity counts (52 vs. 56), indicating that it is uncertain that a student will successfully complete the programming activity or not. The *EC cluster* has a higher number of completions. Thus, progression trajectories assigned to the *EC cluster* are more likely to finish successfully for a given programming activity. Even though there are a small number of incomplete programming activities in the *EC cluster*, these students progressed through the programming activities fairly quickly and were close to the expert solutions.

Next, we compared how these clusters are distributed across programming activities. Figure 4 shows that early programming activities have a higher number of progression trajectories that belong to the *EC cluster* whereas the later programming activities have an increased number of progression trajectories in the *EQ cluster*. This is expected, as many students will make quick progress through earlier programming activities due to their low difficulty and the later programming activities have a higher rate of incompletion. Programming Activity 5 has the highest occurrences of the *HP cluster*, suggesting that students spend a significantly long time in Programming Activity 5.

To examine the differences in student actions across clusters, we investigated the frequency of requesting hints and restarting the current programming activity from scratch. We also performed a pairwise Tukey honestly significant difference test to verify statistically significant results. As shown in Figure 5, progression trajectories in the *HP cluster* requests significantly more hints (M=7.18, SD=8.37, min=0, max=49) than those in the *EQ cluster* (M=0.81, SD=1.78, min=0, max=10, p<0.001) and the *EC cluster* (M=1.11, SD=2.63, min=0, max=22, p<0.001). This can be explained as progression trajectories in the *HP cluster* spend a significantly longer time progressing through the programming activities (Figure 2). Trajectories in the *EQ cluster* and the *EC cluster* either completes the programming activity shortly or leaves the programming activity unfinished without sufficient effort, so their low number of hint requests are expected as well. It is also interesting to observe that there is no significant difference between hint requests in the *EQ cluster* and the *EC cluster* (p=0.650).

When progressing through a programming activity, students sometimes remove all blocks from the workspace to start fresh. We describe this behavior as a *restart*. To count the number of *restarts*, we selected a threshold of 10%, meaning whenever similarity percentage falls below 10% and is then followed by an increase, we categorize this as a single *restart*. Considering the number of restarts in progression trajectories across clusters, the *HP cluster* has the highest number of restarts (M=3.42, SD=2.86), whereas the *EQ cluster* (M=0.37, SD=1.41) and the *EC cluster* (M=0.33, SD=0.67) have significantly low numbers of restarts (p<0.001 for both). This is also expected as trajectories in the *HP cluster* show significant effort to finish a programming activity and thus restarts more frequently than other clusters. Progression trajectories that belong to the *EQ cluster* tend to restart less frequently. This suggests that students following these projection trajectories are less willing to start again, leaving the programming activity unfinished. Moderate numbers of restarts are seen in trajectories that belong to the *EC cluster*, implying the tendency to start fresh when stuck.

To understand how these clusters are associated with prior knowledge (i.e., pre-test) and overall performance (i.e., post-test) of

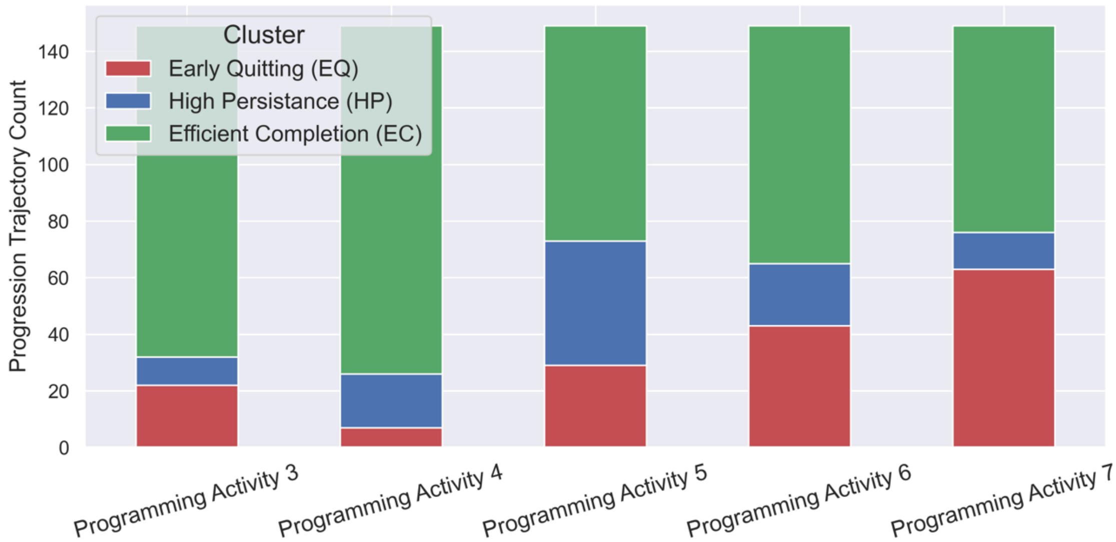

Figure 4: Number of progression trajectories in each cluster for Programming Activity 3, 4, 5, 6, and 7 (from left to right)

each student, we assigned each student to a student group based on their majority association with a particular cluster across all five programming activities. Students assigned to the same cluster in three or more programming activities (out of five) are tagged as having that cluster as a majority. Among 149 students, 98 belong to *Early Completion* major (EC Major), 25 belong to *Early Quitting* major (EQ Major), 6 belong to *High Persistence* major (HP Major), and the remaining 20 have mixed clusters (Mixed) in their progression through the programming activities. Analysis of the pre-test scores using Tukey honestly significant difference test (Figure 5) shows no significant differences between HP Major, EQ Major, and Mixed student groups. But EC Major groups have significantly higher ($p<0.001$) pre-test scores ($M=66.26$, $SD=18.81$) than EQ Major ($M=33.05$, $SD=14.83$) and Mixed ($M=47.58$, $SD=23$) student groups. This is expected as students with higher prior knowledge should progress through the programming activities more easily (early completion) than others.

For the post-test scores, there is a significant difference between HP Major and EQ Major ($p=0.014$), between EC Major and Mixed ($p=0.009$), between EQ Major and EC Major ($p<0.001$), and between EQ Major and Mixed ($p=0.028$) student groups. From this, it is evident that students progressing through programming activities using specific progression trajectories have strong associations with their post-test performance. The EC Major student group has a higher post-test score ($M=69.53$, $SD=21.46$) than EQ Major student groups ($M=35.97$, $SD=16.08$) and Mixed student groups ($M=53.33$, $SD=23.08$) but have no significant difference with the HP Major student group ($M=64.58$, $SD=6.7$). In other words, students whose majority of the progression trajectories belong to the *HP clusters* or the *EC clusters* have the highest post-test performance, suggesting significant learning outcomes for these two groups, whereas the EQ Major student group has the lowest post-test performance compared to the other groups. The Mixed student group, having no major association with any particular progression trajectory cluster, demonstrates moderate post-test performance with high standard deviation.

A notable observation here is that EC Major students have almost identical pre-test scores ($M=66.26$, $SD=18.81$) and post-test scores ($M=69.53$, $SD=21.46$), suggesting no significant learning (*paired t test* $p=0.051$) was observed. Same is also true for EQ Major students (mean pre-test score of 33.05, mean post-test score of 35.97, *paired t test* $p=0.172$). But the Mixed student group and the HP Major student group have noteworthy differences (*paired t test* $p=0.042$ and $p=0.029$ respectively) between their pre-test and post-test score.

With regard to the second research question (RQ2), the analysis shows that these three clusters have significantly different characteristics in programming activity completion, number of hint requests, and number of restarts. Students majorly associated with these clusters also show significantly different pre-test and post-test scores.

4.3 Early Prediction of Next-Activity Cluster

Investigating the behavior of individual students across programming activities, we found that 34 (23%) students did not change clusters across all five programming activities. Among 596 programming activity transitions (no transition going into Programming Activity 3), 65% of transitions remained in the same cluster in consecutive programming activities. More granular details can be found in Table 2

To understand how students transition between clusters across all five programming activities, we conducted sequential pattern mining using the prefixspan[2] Python package on the clusters [16]. Table 3 depicts the five most frequent patterns when progressing through the programming activities. As we can see, 33 (22%) students remained in the *EC cluster* across all five programming

[2] https://pypi.org/project/prefixspan/

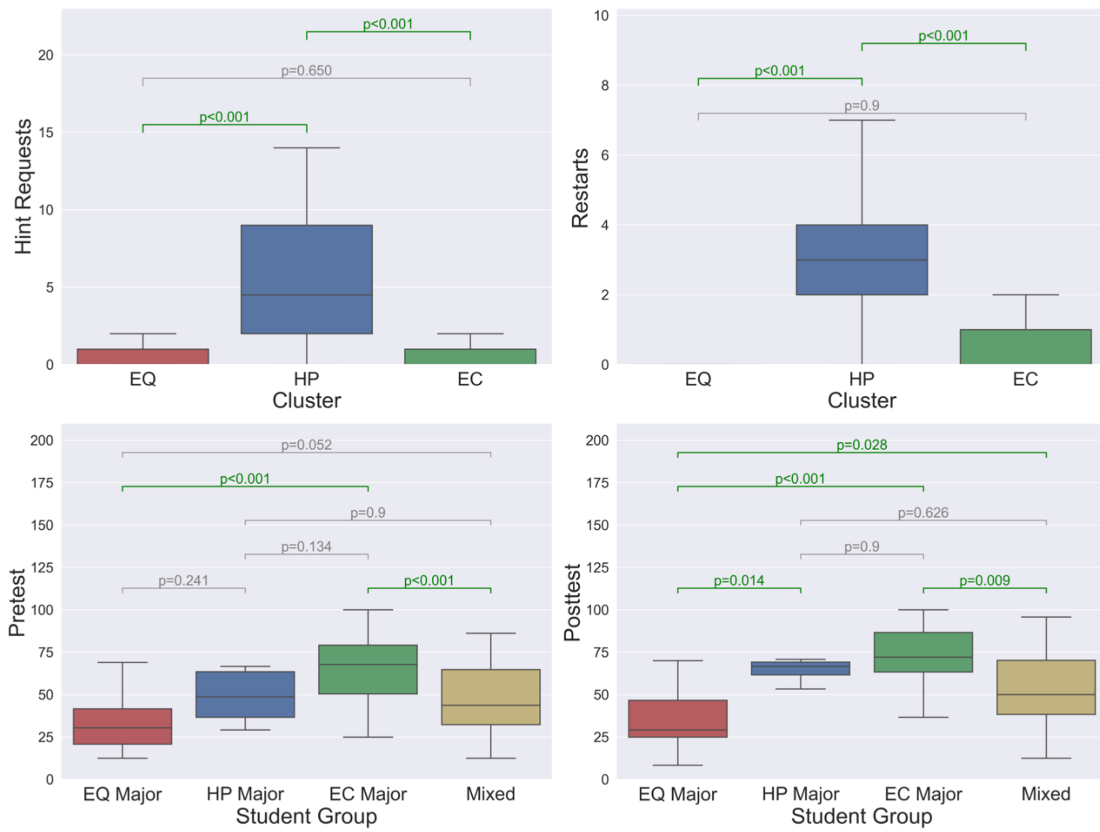

Figure 5: The top row shows the number of Hint Requests (top-left) and the number of Restarts (top-right) across three clusters. The bottom row shows the Pre-test (bottom-left) and Post-test (bottom-right) scores across all four student groups.

Table 2: Students remaining in the same cluster in consecutive programming activity

Programming Activity Transition	Students with Same Cluster
3 to 4	107 (71.81%)
4 to 5	84 (56.38%)
5 to 6	97 (61.1%)
6 to 7	100 (67.11%)
Total	388 (65.1%)

Table 3: Top five patterns of cluster change across all five programming activities

Number of Students	Cluster Change from Activity 3 to 7
33 (22.15%)	EC -> EC -> EC -> EC -> EC
12 (8.05%)	EC -> EC -> HP -> EC -> EC
12 (8.05%)	EC -> EC -> EQ -> EQ -> EQ
8 (5.37%)	EC -> EC -> EC -> EC -> EQ
6 (4.03%)	EC -> EC -> HP -> EQ -> EQ

activities. Another 12 students started their progression trajectories in the *EC cluster* and changed to *EQ cluster* after Programming Activity 4, implying that these students might have lost interest or lack the prior knowledge to complete more challenging programming activities. A total of 12 other students remain in the *EC cluster* and only transitioned to the *HP cluster* once in Programming Activity 5. These students appeared to expend significant effort when first faced with a difficult programming activity and learned to quickly progress through the rest.

On average, two-thirds (65%) of the students do not change their progression trajectory patterns (i.e., remain in the same cluster) in consecutive programming activities. This is noteworthy and can be used to accurately predict the behavior of students in consecutive programming activities. To demonstrate this, we designed two evaluations to do early prediction of students' future behavior. The first evaluation (TASK Cluster Change) was a binary classification task to predict if a student would change her progression trajectory pattern in the next programming activity or not (binary class). The second evaluation (TASK Next Cluster) was a multi-class classification task to predict the progression trajectory (i.e., cluster) of a student in the next programming activity. For both of these evaluations, we trained four commonly used off-the-shelf classifiers using scikit-learn [37]: random forests (RF), decision

Table 4: (TASK Cluster Change) F1 weighted macro score for predicting if a student will change their current progression trajectory cluster (binary) in the next programming activity

Programming Activity	Base Prev	Base Major	RF	DT	LR	SVC
3 to 4	60%	60%	81%	79%	84%	**87%**
4 to 5	44%	41%	**61%**	60%	56%	**61%**
5 to 6	65%	51%	74%	60%	70%	**78%**
6 to 7	55%	54%	67%	65%	62%	**70%**
All	58%	51%	71%	67%	69%	**72%**

trees (DT), logistic regression (LR), and support vector classification (SVC). We used a total of 30 features to train the models, which included three sets of features: system log, prior performance, and current activity progress. The system log features contain 16 event logs, which are counts of the following actions: *hint request*, *total number of interactions with the environment*, *load last save*, *load previous exercise code*, *next instruction*, *previous instruction*, *run code*, *save workspace*, *change block*, *create block*, *delete block*, *move block*, *number of toolbox interactions*, *create variable*, *delete variable*, and *rename variable*. The prior performance features group includes 9 features: total time spent up to the current moment, total programming activities completed so far, total number of restarts so far (as defined in 4.2), total number of trajectories belonged to each cluster, total workspace cleared per restart so far, total number of clusters changed so far, total cumulative final similarity so far. These features are calculated using the cumulative sum of previous programming activities for a particular student. Lastly, the current activity progress features group includes 5 features: the number of restarts in the current programming activity, current cluster, time spent in current programming activity, is the current programming activity successfully completed, and the final similarity percentage of the current progression trajectory. We standardized all the features to remain consistent across different feature scales. As the distribution of clusters is imbalanced (see 4.1), the only parameter manually tuned was the class_weight balanced (where applicable). The remainder of the hyperparameters were set to the default values.

In the evaluations, we used two baselines. The first baseline, BASE Major, predicts the majority class in both of the classification tasks. The second baseline, BASE Prev, predicts the same class in consecutive programming activities. For TASK Cluster Change, this means carrying over the current cluster changing behavior in the next programming activity, and for TASK Next Cluster, this means carrying over the current cluster in the next programming activity. For TASK Cluster Change, as both of the classes (students who will remain in the same cluster, and students who will not) are equally important to predict, and for TASK Next Cluster, as all three clusters (EQ, HP, and EC) are equally important, we have reported the F1 scores, instead of precision or recall. We calculated F1 weighted macro (average F1 score based on support on each class) to account for class imbalance. For all evaluations, we performed 10-fold cross-validation, repeated 10 times (total of 100 runs for each classifier), and reported the median F1 scores (mean values are affected by outliers [26]).

TASK Cluster Change: As shown in Table 4, when training on *All* programming activity transitions, all classifiers outperform (9 to 14% higher) the best baseline (BASE Prev). Examining each programming activity transitions, we can see that it is easier to predict the cluster change in Programming Activity 3 to 4 (19 to 27% higher than both baselines) and in Programming Activity 4 to 5 (12 to 17% higher than the best baseline), but it is more difficult to predict cluster changes in later programming activity transitions, specifically from Programming Activity 5 to 6 (-5 to 13% higher than best baseline) and from Programming Activity 6 to 7 (7 to 15% higher than best baseline). In all cases, almost all the classifiers significantly outperform the baselines. Also, note that SVC outperforms every other classifier in all cases.

The fact that predicting cluster changes from Programming Activity 5 to 6 is difficult is noteworthy. Looking into the programming activity description (Table 1), Programming Activity 6 introduces loops for the first time. It might be the case that students find this concept hard to learn and struggle. As such, classification based on prior knowledge is not sufficient.

TASK Next Cluster: As shown in Table 5, predicting specific clusters (*EQ, HP,* and *EC*) is a challenging task because the three classes are highly imbalanced. For *All* programming activity transitions, the F1 scores of the classifiers are ranging between 62% (DT) to 68% (RF) whereas the Base Prev, which is the best baseline, has a score of 64%. Improvements can be seen when predicting clusters for Programming Activity 5 using data from Programming Activity 4 (3 to 11% improvement over best baseline). Apart from DT predicting clusters in Programming Activity 7, all the classifiers perform somewhat similarly to the best baselines, if not better, in all cases. Investigating further, we found that most of the time the clusters are wrongly predicted as *EC clusters*. For example, for random forest classifiers, the *HP cluster* gets misclassified as *EC cluster 76%* of the time and *EQ cluster* gets misclassified as *EC cluster 25%* of the time. Similar results are seen across other classifiers. This can be explained by the fact that the classes are mostly imbalanced, having 60% of the progression trajectories belong to the *EC cluster* and only 16% belong to *HP cluster*. This makes the prediction of the classifiers biased towards the *EC cluster*. Also note that *HP cluster*, being the uncertain cluster (Figure 3), can go either way and is predicted towards the majority class (*EC cluster*).

With regard to the final research question (RQ3), the two evaluations suggest that we can accurately predict students' cluster changing behaviors in consecutive programming activities, but predicting the exact cluster is somewhat difficult due to imbalanced data.

Table 5: (TASK Next Cluster) F1 weighted macro score for predicting the progression trajectory cluster (EQ, HP, or EC) of a student in the next programming activity

Programming Activity	Base Prev	Base Major	RF	DT	LR	SVC
3 to 4	74%	75%	77%	72%	**80%**	76%
4 to 5	50%	34%	59%	53%	**61%**	59%
5 to 6	**67%**	41%	65%	65%	63%	**67%**
6 to 7	**69%**	32%	68%	54%	62%	61%
All	64%	45%	**68%**	62%	66%	65%

5 DISCUSSION

Understanding students' progression trajectories can provide insight to guide the design of more effective adaptive learning environments. Our analysis found three generic progression trajectory patterns when students progress through individual programming activities in block-based programming environments. Similar patterns have also been seen in earlier studies across different programming environments [22, 23, 39], but unlike this work, these studies found student-specific patterns, rather than their behavior in individual programming activities. However, our findings echo the findings of previous studies. For example, students who primarily belong to the *EQ cluster* (giving up early without spending much time or effort) across multiple programming activities would likely be identified as *stoppers* or *quitters* whereas students who primarily belong to the *HP cluster* (spent significantly more time to trying to solve a programming activity) across multiple programming activities would likely be identified as *tinkerers* or *approachers*. Similarly, student progression trajectories that primarily belong to the *EC cluster* could be identified as *movers, knowers,* or *solvers*. Overall, these student progression trajectory clusters offer a fine-grained understanding of student behaviors in individual programming activities and have significant potential to improve learning environments in future.

The analysis in this paper also shows these clusters to have significantly different characteristics across multiple dimensions. Their distributions across programming activities show that earlier programming activities with lower difficulties have lower number of progression trajectories in *EQ cluster* whereas later programming activities with higher difficulties have increasingly higher number of progression trajectories in *EQ clusters*. This implies that students tend to early-quit more often as difficulty increases. Similarly, student progression trajectories in the *HP cluster* show significantly more hint requests and restarts. Thus, students primarily belonging to the *HP cluster* across multiple programming activities (HP Major) are more persistent in the learning process and their outcomes are reflected by their positive changes in distributions for pre-test and post-test scores. As such, it can be seen that difficult activities need better support or timely motivation to reduce early quitting while ensuring higher persistence for better learning outcomes.

Thus, predicting these clusters early for individual students can inform the design of adaptive support in learning environments. For example, by identifying a student who will change their current behavior in the upcoming activity, the environment can adapt its support system to provide more or less student-adaptive hints or feedback accordingly. Environments can also utilize these predictive future behavior patterns to motivate students to increase their engagement or to discourage early quitting.

6 CONCLUSION

Student modeling for block-based programming environments can inform student-adaptive learning experiences with feedback and guidance that is tailored to individual students to help them master programming concepts. However, devising student models for block-based programming poses significant challenges because students can pursue a multiplicity of paths to solutions. To address these issues, we have introduced progression trajectory-based student models for novice block-based programming environments. Progression trajectory-based student models utilize a time-series representation that employs code analysis to incrementally compare the intermediate programming steps of students' evolving programs to expert programs. We investigated progression trajectory-based student models with student programming interactions in an introductory college-level computer science course. Analysis of the models found that 1) there are three distinct patterns among such trajectories, *Early Quitting (EQ), High Persistence (HP),* and *Efficient Completion (EC)*, which exhibit significantly different characteristics across multiple dimensions, and 2) the models can be used to accurately predict learning behaviors.

The results suggest several promising directions for future work. First, it will be important to consider different embeddings techniques of intermediate code states. For example, vector-based embeddings of the intermediate code states can reflect more granular and structured understanding of students' program evolution. Second, future work should investigate approaches to further improving the predictability of next programming activity patterns by considering different feature sets including incoming knowledge, perceived skills, and program code. To this end, devising feature extraction techniques appears promising. Third, it will be instructive to investigate integrating trajectory-based student models into adaptive learning environments to understand how they can most effectively contribute to student-adaptivity to improve student learning.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation through grants DUE-1626235 and DUE-1625908. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Jordan Barria-Pineda, Julio Guerra-Hollstein, and Peter Brusilovsky. 2018. A fine-grained open learner model for an introductory programming course. In *Proceedings of the 26th Conference on User Modeling, Adaptation and Personalization*, 53–61.

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

[3] Paulo Blikstein. 2011. Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, 110–116.

[4] 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 Learning Science* 23, 4 (2014), 561–599.

[5] Eric Brill and Robert C Moore. 2000. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*, 286–293.

[6] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*, 53–62.

[7] Nicholas Diana, Michael Eagle, John Stamper, Shuchi Grover, Marie Bienkowski, and Satabdi Basu. 2017. An instructor dashboard for real-time analytics in interactive programming assignments. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*, 272–279.

[8] Nicholas Diana, Michael Eagle, John Stamper, Shuchi Grover, Marie Bienkowski, and Satabdi Basu. 2018. Data-driven generation of rubric criteria from an educational programming environment. In *Proceedings of the 8th International Conference on Learning Analytics and Knowledge*, 16–20.

[9] Andrew Emerson, Michael Geden, Andy Smith, Eric Wiebe, Bradford Mott, Kristy Elizabeth Boyer, and James Lester. 2020. Predictive Student Modeling in Block-Based Programming Environments with Bayesian Hierarchical Models. In *Proceedings of the 28th ACM Conference on User Modeling, Adaptation and Personalization*, 62–70.

[10] Andrew Emerson, Andy Smith, Fernando J. Rodriguez, Eric N. Wiebe, Bradford W. Mott, Kristy Elizabeth Boyer, and James C. Lester. 2020. Cluster-based analysis of novice coding misconceptions in block-based programming. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (2020), 825–831. DOI:https://doi.org/10.1145/3328778.3366924

[11] Andrew Emerson, Andy Smith, Cody Smith, Fernando Rodríguez, Eric Wiebe, Bradford Mott, Kristy Boyer, and James Lester. 2019. Predicting Early and Often: Predictive Student Modeling for Block-Based Programming Environments. In *Proceedings of the 12th International Conferece on Educational Data Mining* (2019), 39–48.

[12] Anthony Estey, Hieke Keuning, and Yvonne Coady. 2017. Automatically classifying students in need of support by detecting changes in programming behaviour. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 189–194.

[13] Neil Fraser. 2013. Blockly: A visual programming editor. *URL https//code. google. com/p/blockly* 42, (2013).

[14] Dan Garcia, Brian Harvey, and Tiffany Barnes. 2015. The beauty and joy of computing. *ACM Inroads* 6, 4 (2015), 71–79.

[15] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interactions (TOCHI)* 22, 2 (2015), 1–35.

[16] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th International Conference on Data Engineering*, IEEE Washington, DC, USA, 215–224.

[17] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 2012. How do students solve parsons programming problems? an analysis of interaction traces. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, 119–126.

[18] Nathan Henderson, Vikram Kumaran, Wookhee Min, Bradford Mott, Ziwei Wu, Danielle Boulden, Trudi Lord, Frieda Reichsman, Chad Dorsey, and Eric Wiebe. 2020. Enhancing Student Competency Models for Game-Based Learning with a Hybrid Stealth Assessment Framework. In *Proceedings of the 13th International Conference on Educational Data Mining* (2020), 93-103.

[19] Roya Hosseini, Peter Brusilovsky, Michael Yudelson, and Arto Hellas. 2017. Stereotype modeling for Problem-Solving performance predictions in MOOCs and traditional courses. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, 76–84.

[20] Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Exploring problem solving paths in a Java programming course. In *Psychology of Programming Interest Group Annual Conference 2014*, University of Pittsburgh, 65.

[21] 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.

[22] Bo Jiang, Wei Zhao, Nuan Zhang, and Feiyue Qiu. 2019. Programming trajectories analytics in block-based programming language learning. *Interactive Learning Environments* 4820, (2019), 1–14. DOI:https://doi.org/10.1080/10494820.2019.1643741

[23] Shamya Karumbaiah, Ryan S Baker, and Valerie Shute. 2018. Predicting Quitting in Students Playing a Learning Game. In *Proceedings of the 11th International Conference on Educational Data Mining* (2018).

[24] Trupti M Kodinariya and Prashant R Makwana. 2013. Review on determining number of Cluster in K-Means Clustering. *International Journal* 1, 6 (2013), 90–95.

[25] Abe Leite and Saúl A Blanco. 2020. Effects of Human vs. Automatic Feedback on Students' Understanding of AI Concepts and Programming Style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 44–50.

[26] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. 2013. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology* 49, 4 (2013), 764–766.

[27] Zitao Liu, Songfan Yang, Jiliang Tang, Neil Heffernan, and Rose Luckin. 2020. Recent advances in multimodal educational data mining in k-12 education. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 3549–3550.

[28] Yuetian Luo and Zachary A. Pardos. 2018. Diagnosing University student subject proficiency and predicting degree completion in vector space. *32nd AAAI Conference on Artificial Intelligence* 32, 1, (2018), 7920–7927.

[29] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 55–106.

[30] Mehak Maniktala, Christa Cody, Amy Isvik, Nicholas Lytle, Min Chi, and Tiffany Barnes. 2020. Extending the Hint Factory for the assistance dilemma: A novel, data-driven HelpNeed Predictor for proactive problem-solving help. *arXiv Prepr. arXiv2010.04124* (2020).

[31] Victor J Marin, Tobin Pereira, Srinivas Sridharan, and Carlos R Rivero. 2017. Automated personalized feedback in introductory Java programming MOOCs. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, IEEE, 1259–1270.

[32] Wookhee Min, Megan H Frankosky, Bradford W Mott, Jonathan P Rowe, Eric Wiebe, Kristy Elizabeth Boyer, and James C Lester. 2015. DeepStealth: leveraging deep learning models for stealth assessment in game-based learning environments. In *Proceedings of the 17th International Conference on Artificial Intelligence in Education*, Springer, 277–286.

[33] Wookhee Min, Bradford Mott, Jonathan Rowe, Barry Liu, and James Lester. 2016. Player goal recognition in open-world digital games with long short-term memory networks. In *Porceedings of the 25th International Joint Conference on Artificial Intelligence* (2016), 2590–2596.

[34] Benjamin Paaßen, Claudio Gallicchio, Alessio Micheli, and Barbara Hammer. 2018. Tree edit distance learning via adaptive symbol embeddings. In *Proceeding of the 35th International Conference on Machine Learning*, PMLR, 3976–3985.

[35] Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2017. The continuous hint factory-providing hints in vast and sparsely populated edit distance spaces. *arXiv Prepr. arXiv1708.06564* (2017).

[36] Zachary A Pardos, Steven Tang, Daniel Davis, and Christopher Vu Le. 2017. Enabling real-time adaptivity in MOOCs with a personalized next-step recommendation framework. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, 23–32.

[37] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, and Vincent Dubourg. 2011. Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research* 12, (2011), 2825–2830.

[38] Radek Pelánek and Jiří Řihák. 2017. Experimental analysis of mastery learning criteria. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, 156–163.

[39] D N Perkins and Fay Martin. 1986. Fragile knowledge and neglected strategies in novice programmers. In *at Empirical Studies of Programmers, 1st Workshop, Washington, DC*, 213–229.

[40] 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.

[41] Thomas W Price and Tiffany Barnes. 2015. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, 91–99.

[42] Thomas W Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating Data-Driven Hints for Open-Ended Programming. In *Proceedings of the 9th International Conference on Educational Data Mining* (2016), 191-198.

[43] 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.

[44] Thomas W Price, Rui Zhi, and Tiffany Barnes. 2017. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *Proceedings of the 18th International Conference on Artificial Intelligence in Education*, Springer, 311–322.

[45] Alexander Renkl and Robert K Atkinson. 2003. Structuring the transition from example study to problem solving in cognitive skill acquisition: A cognitive load perspective. *Educational Psychologist* 38, 1 (2003), 15–22.

[46] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, and Brian Silverman. 2009. Scratch: programming for all. *Communication of the ACM* 52, 11 (2009), 60–67.

[47] Kelly Rivers, Erik Harpstead, and Kenneth R Koedinger. 2016. Learning curve analysis for programming: Which concepts do students struggle with? In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 143–151.

[48] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64.

[49] Fernando J Rodríguez, Kimberly Michelle Price, Joseph Isaac, Kristy Elizabeth Boyer, and Christina Gardner-McCune. 2017. How block categories affect learner satisfaction with a block-based programming interface. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 201–205.

[50] Pavel Senin. 2008. Dynamic time warping algorithm review. *Information and Computer Science Department University of Hawaii at Manoa Honolulu, USA* 855, 1–23 (2008), 40.

[51] Melissa Stanger and Emmie Martin. 2016. The 50 Best Computer-Science and Engineering Schools in America, 2015.

[52] Giovanni Maria Troiano, Sam Snodgrass, Erinç Argımak, Gregorio Robles, Gillian Smith, Michael Cassidy, Eli Tucker-Raymond, Gillian Puttick, and Casper Harteveld. 2019. Is my game OK Dr. Scratch? Exploring programming and computational thinking development via metrics in student-designed serious games for STEM. In *Proceedings of the 18th ACM International Conference on Interaction Design and Children*, 208–219.

[53] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. 2017. Learning to represent student knowledge on programming exercises using deep learning. In *Proceedings of the 10th International Conference on Educational Data Mining, EDM 2017* (2017), 324–329.

[54] Christiane Gresse Von Wangenheim, Jean C R Hauck, Matheus Faustino Demetrio, Rafael Pelle, Nathalia da Cruz Alves, Heliziane Barbosa, and Luiz Felipe Azevedo. 2018. CodeMaster–Automatic Assessment and Grading of App Inventor and Snap! Programs. *Informatics in Education.* 17, 1 (2018), 117–150.

[55] Christopher Watson and Frederick W B Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 39–44.

[56] David Weintrop and Uri Wilensky. 2017. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computer Education* 18, 1 (2017), 1–25.

[57] Joseph B Wiggins, Fahmid M Fahid, Andrew Emerson, Madeline Hinckle, Andy Smith, Kristy Elizabeth Boyer, Bradford Mott, Eric Wiebe, and James Lester. 2021. Exploring Novice Programmers' Hint Requests in an Intelligent Block-Based Coding Environment. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (2021), 52-58.

[58] Jeannette M Wing. 2014. Computational thinking benefits society. *40th Anniversary Blog on Social Issues in Computing* (2014), 26.

[59] Rui Zhi, Thomas W Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. 2019. Exploring the impact of worked examples in a novice programming environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 98–104.