

EARLY PREDICTION OF STUDENT PROGRAMMING PROCESSES IN BLOCK-
BASED LEARNING ENVIRONMENTS

by

Punarva Vyas

Submitted in partial fulfilment of the requirements
for the degree of Master of Applied Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December, 2020

© Copyright by Punarva Vyas, 2020

TABLE OF CONTENTS

LIST OF TABLES	iii
LIST OF FIGURES	iv
ABSTRACT.....	v
ACKNOWLEDGEMENTS	vi
CHAPTER 1 INTRODUCTION	1
1.1 PROGRAMMING ANALYTICS.....	4
CHAPTER 2 METHODS	8
2.1 DATASET.....	8
2.2 DATA PREPROCESSING	9
2.3 SEGMENTATION WINDOW	12
2.4 FEATURE EXTRACTION	13
2.5 EVALUATION METHOD.....	14
CHAPTER 3 RESULTS	15
3.1 FEATURE EXTRACTION	15
3.2 SEGMENTATION WINDOW	17
CHAPTER 4 DISCUSSION.....	21
4.1 MODEL EVALUATION.....	21
4.2 LIMITATION	23
4.3 CONCLUSION	24
BIBLIOGRAPHY	26
APPENDIX A. EARLY PREDICTION MODEL.....	29

LIST OF TABLES

Table 1. Encoding approaches used for parsing and level of granularity for each edit in a windowed segment.....	15
Table 2. Goodness of fit metric estimates on resampling by type of encoding for parsing and level of granularity	16
Table 3. Window sizes used for 3-gram semantic encoding and order of edit in each windowed segment.....	18
Table 4. Goodness of fit metric estimates on resampling by type of segmentation window size and step	19
Table 5. Confusion matrix for the early prediction of student-produced solution correctness at the last execution.....	22

LIST OF FIGURES

Figure 1. Snapshot of the columns in the dataset.....	8
Figure 2. The RoboMission programming task. From left to right, (1) the problem description, (2) the given blocks, (c) the best solution, and (d) the most common error solution in program execution	10
Figure 3. Solution state representation in RoboMission.	11

ABSTRACT

This study examined the impact of different feature extraction and segmentation window methods on early prediction of students' learning to program with block-based learning environments. Estimates for the goodness of fit metrics were obtained for metrics sensitive to the cost of misclassification, including recall, F1, and lift metrics. The first resampling experiment compared semantic and character encoding to extract features that characterized student-produced programs while varying the granularity for contiguous sequences. The results showed that compared to the character-encoding approach, the best fitting models were obtained through longer sequences of semantic information to represent statements and commands. In the second experiment, the size and step of the segmentation window to segment series of student edits to their program was compared. Namely, the concatenation of 5 or 10 contiguous edits and partitioning the time series at the 5th, 15th, and 25th edit during the learning session. The results showed that the best performing models were obtained when segmenting the edit sequences earlier during the learning session while taking into account a larger number of edits. These results support the deployment of an early prediction model to differentiate solution correctness on the basis of edits made to the program in real-time, enabling block-based learning environments to personalize instruction by providing partially complete solutions during task performance as a means to improve learning.

ACKNOWLEDGEMENTS

A special thanks to my supervisor, Dr. Eric Poitras, for his advice and guidance throughout the completion of this project. In addition, I would like to thank Tomáš Effenberger who made the dataset publicly available that made the research project feasible.

CHAPTER 1 INTRODUCTION

Computational thinking can be defined as a problem-solving strategy that involves expressing problems and their solutions such that computer could execute. It requires strong mental and thinking abilities to provide computers with the understanding and solutions of real-world problems. The educational system has evolved from fact-based learning to problem solving based learning. Students are inclining towards innovation and creativity. Thus, a large influx of students is seen in computer programming. Computational thinking is the main foundation in computer programming. Students are required to think as they are computers. Blockly is a block-based programming game which presents its user with challenging tasks that enhances the users thinking and cognitive skills. Blockly is developed such that the users have to be more focussed on logics rather than learning the syntax of the problems [1-2].

Since the term was first introduced by Seymour Papert [3] in 1981, computational thinking education has been attracting a lot of interest. The assumption is that logical thinking and problem-solving skills are fundamental to how computer scientists think and should be taught to students in a manner similar to reading, writing, and arithmetic. In contrast to learning how to program, the meaning of the term has emerged from an emphasis on fundamental analytical skills used to understand and solve problems more effectively by drawing on foundational concepts and technologies from the computer sciences [4]. Programming, much like learning how to write, is a means of expression that enables students to develop novel understanding such as decomposing a problem into a sequence of steps, formulating precise instructions, and when appropriate, using

conditional instructions or repetition of groups of instructions. Drawing on this definition of computational thinking, an algorithm for instance is introduced to students as a well-defined procedure that may be repeatedly applied to transform an input into a desired output. Students thinking computationally learn to approach real-world problems by formulating solutions based on algorithms and automation and leveraging abstractions to represent changing conditions. These skills are increasingly relevant to many disciplines beyond the computer sciences and have led to sustained efforts from the education community to gain deeper insights into the developmental understanding of students [5-6].

Formative assessment is process of monitoring student learning and providing them with feedback that is used by teachers to improve the quality of education that they are imparting. Providing the instantaneous formative assessment in today's world where learning has shifted to online is bit of a challenge. It consumes a lot of time and providing students with the immediate feedback becomes very difficult. Students who are motivated by earning grades and marks might not be as motivated if not provided with regular assessments. Developing a computer program which provides accurate and immediate feedback can prove to be very useful. Program analysis of the student performance can be carried out and the next task can be suggested based on it [7].

Investigating student knowledge of specific concepts is a continuing concern within the fields of educational mining and learning analytics [8-9]. A recent systematic literature review outlined several different methodological approaches to automate the collection of data in regard to student programming behaviour, ranging from individual keystrokes to

complete assignment submissions [10]. Student programming processes are well-suited towards educational data mining analyses in that it pertains to meaningful student-software interaction data over the span of multiple learning sessions, including mappings between problem steps and knowledge components as well as student demographic, pre post-test assessment, and survey responses [11]. Interactions between students and programming environments are logged using tools such as automated grading systems, online integrated development environments, version control systems, and other software applications with key logging capabilities. Students perform programming tasks while editing source code, compiling, and verifying solutions, as well as submitting their work to receive feedback. In doing so, sequence of program snapshots may be collected and analyzed in real-time for the purposes of tailoring instruction to the specific needs of different students.

This study is therefore set out to assess the effect of early classification of student produced programs, and continually monitoring edits as indicators of common misconceptions. The rationale is that classifying the time series data as early as possible allows the system to deliver timely support to students who are struggling with solving a problem. Criteria for evaluating the proposed modelling approach were as follows: (1) the classifier should be tuned to assert the earliest time of reliable classification as a means to ensure effective intervention from an instructional perspective; (2) the classifier should be sensitive to the different value of false positive and false negative classification errors since failure to intervene when it is necessary to do so is less desirable than the alternative outcome; and (3) the features that serve as input parameters as well as the model representation should be interpretable as a means to differentiate instruction

provided to students. To assess whether and how the early classifier outperforms a classifier that relies on the entire edit sequence as baseline, the data-mined student models are evaluated using student-level cross-validation while adding progressively larger edit sequences. Features are also engineered using either contiguous or hierarchical n-gram extraction approaches for program representation. Finally, the lift metric is calculated at each student learning session as a goodness of fit estimate for model performance. Cross-validation shows that this model should be effective for other students using the same task with RoboMission, a block-based programming learning environment for novice programmers. The following section describes the research background that informs the algorithm for early detection of student performance.

1.1 PROGRAMMING ANALYTICS

RoboMission is a web-based learning environment that is designed to support and engage novice programmers in gaining deeper understanding of the relationship between parts of the code and the overall purpose of a program [12]. Students rely on a visual programming editor that represents features of a programming language as interlocking blocks, similar to other block-based approaches to visual programming such as Scratch [13], Snap! [14], and Blockly [15]. The underlying assumption for visual programming editors is to help structure the learning task by enabling students to deal with more content and skill demands than they could otherwise handle. Students learn how to trace code by looking at a set of interlocking blocks and predicting state changes and outputs through the compilation and execution of a program. Each program is intended to solve a problem in the context of a space-themed grid world; for example, students may create a

program that would lead a spaceship to the last row while avoiding obstacles. Once the program begins execution, students can evaluate each piece of code in relation to the program's purpose without being required to compose syntactically correct code. In doing so, abstract syntax tree elements serve as building blocks or puzzle pieces, allowing students to easily add, edit, and assemble them to form programs without the need to evaluate syntactic elements for errors [16].

The affordances and drawbacks of traditional text-based interfaces compared to block-based, or blended approaches for instruction with novice programmers have been subject to debate within the computer science education community [17-18]. Block-based interfaces have been shown to facilitate instruction by being more legible, easing the burden to recall commands or syntax, as well as typing commands [19-21]. The drawback however is that students may engage in unfavorable habits of programming that include incorrect usages of programming structures and series of fine-grained commands rather than abstractions such as loops [22]. In a comparison of both environments, Weintrop and Wilensky [18] has shown that students perform better on block-based questions related to conditional logic, function calls, and definite loops. However, no differences were found with the isomorphic text alternative in regard to variables, indefinite loops, and program comprehension questions. Weintrop and Wilensky [23] reported improved scores on assessment in high school classrooms, with students using block-based interfaces reporting greater learning gains and higher levels of interest in future computing courses, while those using the text-based interface viewed their experience as more similar to what professional programmers do and as more effective in improving their programming ability. Price and Barnes [24] have also shown

that students complete activities in less amount of time than in isomorphic text environments.

A key aspect in the evaluation of intelligent tutoring systems such as RoboMission includes computational analyses of learner-created programs, regardless of the type of interface. The rationale is to enable the system to tailor instruction to the specific needs of different learners, for instance, by progressively fading block-based in favor of text-based representations with students at various proficiency levels. VanLehn [25] distinguishes between systems that adapt instruction once for each task, where a task consists of solving a multi-step problem in RoboMission, from those that give feedback and hints on each step. Thus far, previous studies have evaluated computational models that enable the system to select the next task that is most appropriate for the student. Effenberger et al. [26] predicted task performance using logistic regression to fit learning curves, where skills are assumed to vary in their difficulty to learn by students over a series of practice opportunities. Rubio [27] outlined a method to estimate the probability of the student successfully completing the next exercise based on the programming trajectory of the student, leveraging patterns extracted from several partial solutions before solving the previous task. Few studies have investigated modeling approaches in terms of giving feedback and hints on each step in any systematic way. The latter point has been examined by Effenberger et al. [28-29], who argues in favor of more nuanced metrics for performance that relies on additional data, such as response times, hints usage, or specific values of incorrect answers.

Research about adaptation of instruction within block-based learning environments based on classification of programming trajectories has been mostly restricted to prediction of future task performance. This study attempts to show the feasibility of early prediction of the correctness of student-produced programs at the last execution during a learning session as a means to intervene through the use of hints, prompts, or partially complete solution examples. The rationale is to adapt instruction at the level of each step taken to solve a problem, rather than at the level of tasks, where the system may recommend the next problem for students to solve. The approach to empirical research adopted for this study was one of resampling for estimation and goodness-of-fit testing under varying conditions for feature extraction and segmentation window. We repeatedly draw subsamples from the given data set to fit and assess the early prediction model while varying these factors: (1) the type of information extracted from student programs (i.e., character and semantic information encoding); (2) the granularity of information encoded (i.e., 1, 2, and 3 contiguous units of information); and (3) segmentation window size (i.e., 5 or 10 edits) and step (i.e., 5th, 15th or 25th edit) to capture sub-sequences in the time-dependent data to establish the ideal time to intervene early during a learning session. This allows for partitioning of the predictive performance and quantifies the uncertainty of estimation of goodness of fit metrics while controlling for confounding effects used to process time series data of student edits made to programs, and is explained further in the following section.

CHAPTER 2 METHODS

2.1 DATASET

The Blockly Programming Dataset [30] has been used in the past to investigate learner modeling approaches in the context of learning programming in a block-based environment [27-28]. Only a few datasets describing student programming processes are made publicly available to enable others to replicate and reproduce results [for review, see 32]. The dataset was chosen for its large number of tasks that capture different program behavior such as commands, loops, and conditions. A total of 85 tasks are divided into linearly and nominally ordered hierarchical problem sets (9 levels, each with 3 sublevels). The data collected by the system consists of student interactions as well as the program states while editing and executing the task solution.

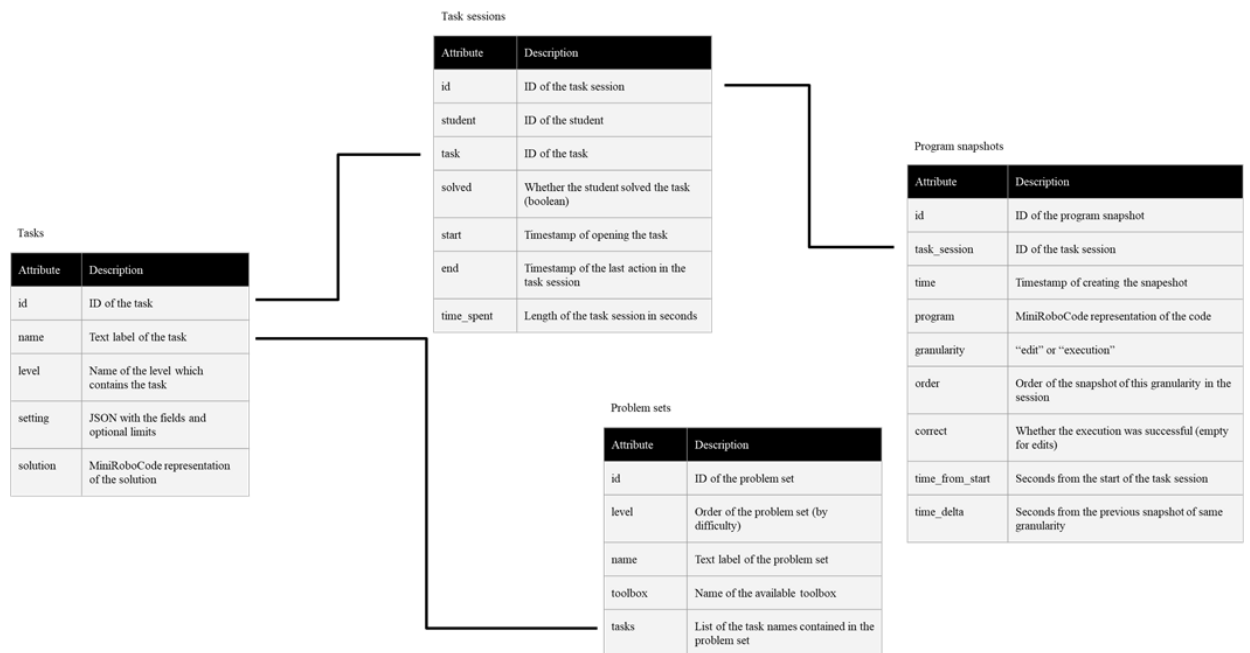


Figure 1. Snapshot of the columns in the dataset

2.2 DATA PREPROCESSING

The programming task is illustrated in Figure 1. The goal of the task is to fly through the space track all the way to the blue line. In doing so, students are given blocks to execute four different commands, either (1) fly ahead, (2) fly left, (3) fly right, or (4) shoot (e.g., shoot to destroy small meteoroids, then move 1 step ahead). Additional blocks include (1) a repeat block that represents a control flow statement for specifying iteration, where a single a variable sets the count parameter value for the number of repetitions (i.e., maximum of 10); and (2) a while block as an alternative statement for iteration, where a conditional statement can be defined through a separate block with two variables, one to declare a comparison operator ($=$ or \neq) and another for a value (i.e., blue, green, red, yellow, or black). The solution must satisfy two constraints. First, the spaceship has to collect all 21 diamonds. Second, students are allowed no more than a total of 8 blocks to create their program. A key condition to the problem is the wormhole (blue symbol in the upper left and lower right area), which enables the spaceship to fly into the square, and then appear in the other square. An indicator shows how many diamonds were already collected and blocks used by the student at each program execution. Students were allowed as many attempts as necessary to find the best solution to the problem. The instructional aim of this task is to facilitate knowledge acquisition regarding nested loops and program sequence, where the first pass of an outer loop triggers an inner loop, which executes to completion. This process is repeated until each outer loop finishes.

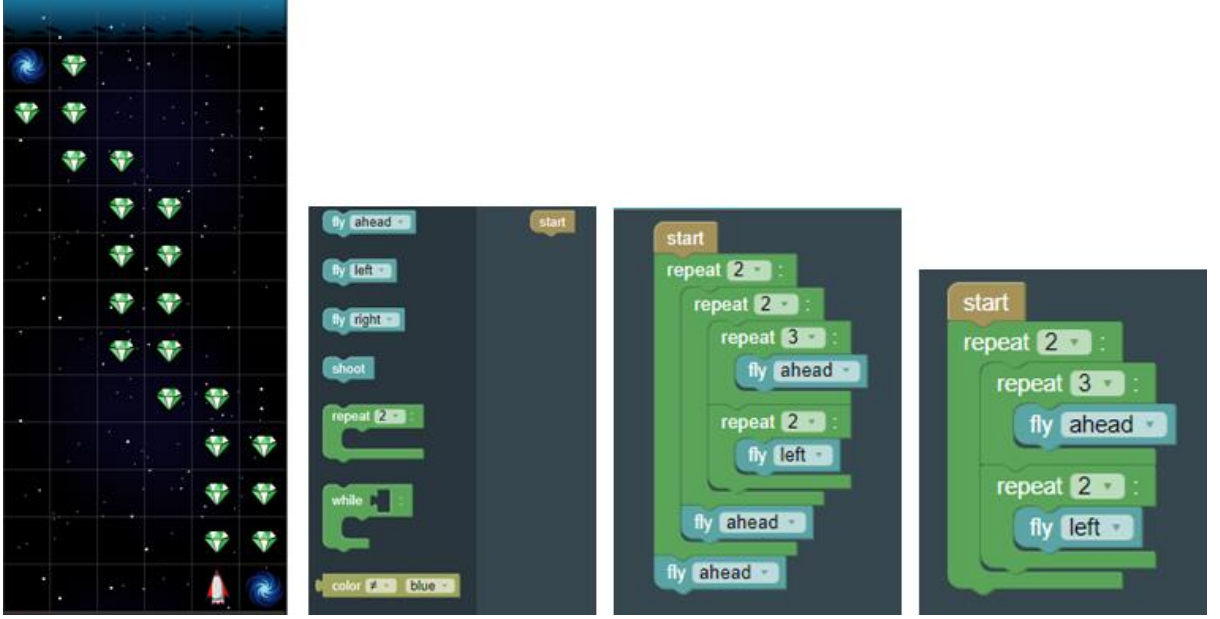


Figure 2. The RoboMission programming task. From left to right, (1) the problem description, (2) the given blocks, (c) the best solution, and (d) the most common error solution in program execution

While a block-based representation of the program is most useful for the students to learn while problem-solving, the system captures the abstract syntax trees representations of these codes for the purposes of translation into two distinct forms. First, Python-like code in text format, which is useful for writing sample solutions. Second, a serialized, compact form of the code in text format that is used for logging snapshots of student programs. Figure 2. shows an example of each type of representation for student programs. Student constructed programs are represented in a serialized form, where each single character of the whole series corresponds to a node in the abstract syntax tree of the program (see Table 1). In addition to program states, student-program interaction data includes identifiers for the task session, timestamps (time, seconds from the start of the session and last attempt at the same level of granularity), its granularity (attempt to edit or execute), the order of attempts made to solve the task, and whether the execution was

successful. Finally, in addition to student-program interaction data, aggregated information such as whether the student solved the task, and the level or difficulty of the task is available to gain insights into the programming process.

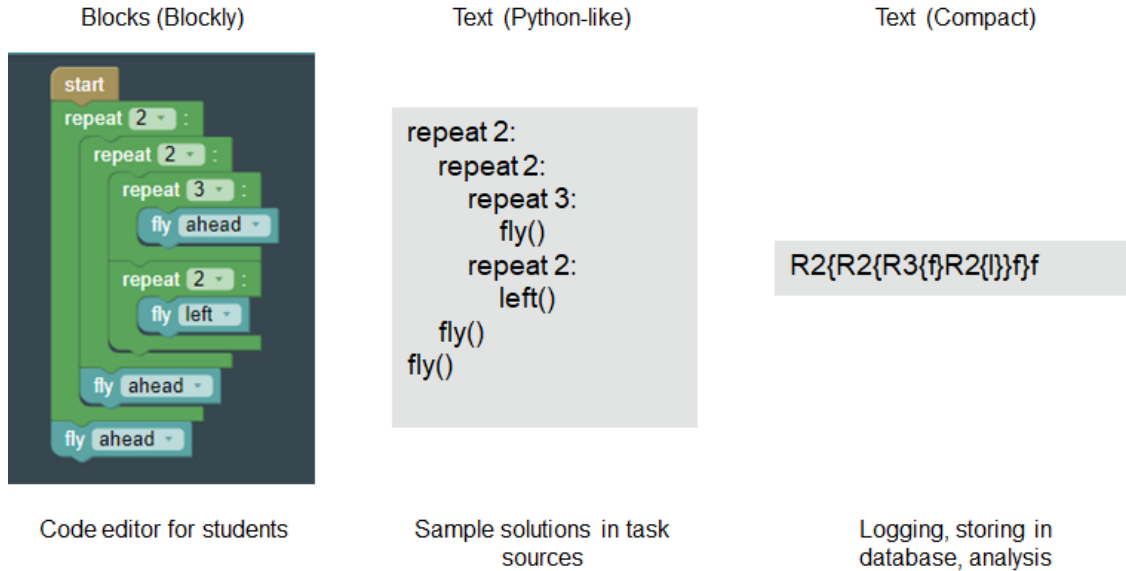


Figure 3. Solution state representation in RoboMission.

When students edited their programs, the log trace data record would capture the compact, text-based representation of the solution state. Programming trajectories can broadly be defined as sequences of program edits and executions generated over time as students make multiple attempts to solve the problem during a single learning session. For the purposes of inferring the correctness of the program at the last execution within a broader trajectory, the examples in the program snapshot table of the Blockly Programming Dataset were filtered for the task under investigation and ordered according to the task session and elapsed time until the beginning of the learning session. Trajectories that did not include at least a single program execution were excluded from further analysis. Furthermore, we removed all examples that corresponded to program executions from within the trajectory, including only examples where students made an

edit to the program. Program states where students deleted all blocks were encoded as “.”. The resulting dataset includes a target attribute (i.e., last execution is correct or not), the order of the edit performed by students (e.g., 1, 2, 3, ...), as well as the program state for the current edit (e.g., $R2\{R2\{R3\{f\}R2\{1\}\}f\}f$). A total of 12,572 examples are labelled as an error in the last execution of the student program, and 13,653 examples are labelled as correct solutions. From the 26,225 examples, a total of 4,791 unique solution states for program edits were recorded in the log data as well as 646 unique task sessions or programming trajectories. The count of edits across programming trajectories ranges from 1 to 351, with an average of 40.7 (SD = 45.97) edits made to the program during the task session.

2.3 SEGMENTATION WINDOW

A sliding window was used to segment programming trajectories into smaller subsequences for the purposes of inferring the correctness of the final task solution. Each subsequence has a fixed length of 5 edits. The non-overlapping window thus captures a set of 5 separate edits, moving from the first set of 5 edits, to the 10th edit in the order within the programming trajectory (i.e., the second set of 5 edit examples), the 15th edit (i.e., the third set of 5 edit examples), and so on. This process is repeated until it reaches the tenth set of 5 edit examples or the 50th edit made within a programming trajectory. Programming trajectories that included less than a total amount of edits covered by the segmentation window were removed from further analysis to control for unequal lengths. Furthermore, the string of text characters that characterize each program state across the set of 5 edits were concatenated as a single example for the purposes of feature

extraction. We compare sliding window steps to segment the programming trajectories as a means to establish the timeliness of predictions made by the classifier, where early predictions that are most accurate are more desirable in terms of guiding instructional interventions.

2.4 FEATURE EXTRACTION

To extract features that represent program edit states that discriminate between correct and incorrect final solutions, we perform character n-gram encoding of the serialized, compact form of the code. We extract n-grams of lengths ranging from 1 to 3 to capture fine-grained ordinal information evident in student programs. This generates all series of characters which have a length of 1 (e.g., R) to 3 (e.g., R2{}), otherwise referred to as uni-, bi-, and tri-grams. Therefore, the ordinal information encodes the placement order of characters regardless of semantics, such as the nature of blocks and their horizontal relationships, and whether these blocks are nested under the same loop statement. The resulting character vector numerically represents each feature in terms of the number of token occurrences (e.g., R occurred 5 times within the string). Akram et al. [33] recently outlined a methodology for encoding abstract syntax trees to capture semantic information stored within the programs, including ordinal and hierarchical properties of the program. While the ordinal information encodes the placement order of blocks that are nested under the same parent node, the hierarchical information encodes what blocks are nested under each other. Both encodings demonstrate the presence of different structures in the compact representation of the concatenated sequence of program edits

and thus, will be compared in terms of deriving valid and meaningful predictions made by the classifier and deriving hints for the purposes of personalized instruction.

2.5 EVALUATION METHOD

An implementation of the C4.5 algorithm was used to partition examples using information gain as the splitting criterion as well as a set of standards, fixed values set for hyperparameters. Following the training and testing phase, the model performance was evaluated using recall, F1, and lift metrics to ascertain the cost of misclassifying positive examples, where students made an error in the final solution for the problem. These metrics are most sensitive to the true positive rate, while accounting for false negatives, false positives, and true negatives, respectively.

CHAPTER 3 RESULTS

3.1 FEATURE EXTRACTION

In order to compare the semantic and character n-gram feature set to infer students' final solution to the problem, this study followed a resampling method. In the resampling procedure, the serialized, compact form of the code that characterized each solution state was either encoded at the level of characters, or parsed for semantic units of information that distinguished between several commands and statements in the block-based program. Furthermore, encoding varied at different levels of granularity to capture combinations of tokens, whether textual characters or meaningful units, ranging from one (uni-), two (bi-), or three (tri-) tokens referred to as n-grams sequences. Then a predictive model was trained to process the feature set as input and differentiate the correctness of the student's final solution to the problem as output. Table 1 shows the feature set used to train the model.

Table 1. Encoding approaches used for parsing and level of granularity for each edit in a windowed segment

Gram	Encoding	
	Character	Semantic
Uni	<u>R</u> 2{ } R2{f} ...	<u>R</u> 2{ } R2{f} ...
Bi	<u>R</u> 2{ } R2{f} ...	<u>R</u> 2{ } R2{f} ...
Tri	<u>R</u> 2{ } R2{f} ...	<u>R</u> 2{ } <u>R</u> 2{f} ...

Note. While semantic encoding characterizes meaningful units of information (i.e., R2, as in repeat two times loop statement), character encoding captures the textual character (i.e., R or the first character in the string segment). An increasing amount of combinations of either characters or semantic information can be encoded, ranging from one, two, or three sequences of characters (i.e., [R, 2, { }]) or meaningful units (i.e., [R2, { }, R2]). A 3-gram sequence would then be represented as either “R_2_{{” for characters

or “R2_{}_R2” for semantic units and weighed on the basis of its frequency of occurrence on a continuous scale ranging from 0, 1, ..., ∞ , where each edit in the windowed segment is separated by “|”.

A sliding window was used to segment programming trajectories into smaller subsequences for the purposes of inferring the correctness of the final task solution. For each iteration of the resampling procedure, the random seed used for partitioning in a 5-fold stratified cross-validation varied ($n = 20$; ranging from 1000, 1105, ... 3000), as well as the step of the non-overlapping fixed-size window to segment 5 edits per time series sequence was varied ($n = 10$; ranging from last 5th edit, 10th edit, ... 50th edit). This was done to control for confounding factors while pairing each sample for statistical comparison. Programming trajectories that included less than a total amount of edits covered by the segmentation window were removed from further analysis to control for unequal lengths. Furthermore, the string of text characters that characterize each program state across the set of 5 edits were concatenated as a single example for the purposes of feature extraction.

Table 2. Goodness of fit metric estimates on resampling by type of encoding for parsing and level of granularity

<i>Metric</i>	<i>Character Encoding</i>			<i>Semantic Encoding</i>		
	<i>Uni-Gram</i>	<i>Bi-Gram</i>	<i>Tri-Gram</i>	<i>Uni-Gram</i>	<i>Bi-Gram</i>	<i>Tri-Gram</i>
Recall	.50	.49	.50	.48	.51	.55*
F1	.49	.49	.49	.48	.49	.51*
Lift	1.09	1.09	1.10	1.10	1.10	1.10

Note. 200 examples per paired samples in each condition, controlling for non-overlapping fixed window step ($n = 10$) and seed for 5-fold stratified cross-validation ($n = 20$) as confounding factors. *Paired-sample two-sided t-test with bonferroni correction, $p < .005$.

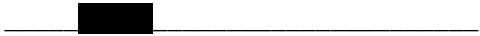
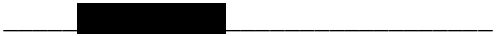


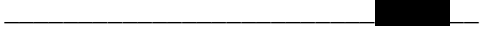
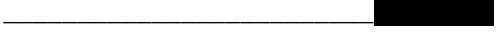
The goodness of fit metric estimates obtained from the resampling procedure by type of encoding for parsing and level of granularity is given in Table 2. Paired-sample t-test comparisons were conducted to ascertain effects using Bonferroni correction to control family wise error rate. As can be seen from the table, the semantic encoding led to better fitting models than character encoding under two conditions. The first condition is for encoding to occur at a higher level of granularity, in that models that relied on tri-gram encoding with semantic information outperformed those that relied on characters. A small effect was obtained in regards to both the recall and F1 scores, $t(199) = 6.22$, $p < .005$, $d = 0.44$; $t(199) = 3.26$, $p < .005$, $d = 0.23$, respectively. The second condition is that the locus of the effect is limited to a specific class of examples, as observed gains in model performance are limited to the recall and F1 metric, but not lift scores. Although the lift metric is sensitive to positive examples given the aims of model evaluation, the metric accounts for true negative rates, in contrast to recall and F1 scores.

3.2 SEGMENTATION WINDOW

For each iteration of the resampling procedure, a different set of seeds used for partitioning in a 5-fold stratified cross-validation were used from the previous experiment ($n = 200$; ranging from 1001, 1011, ... 3001). Window size was manipulated to capture either 5 or 10 contiguous edits in a larger sequence during each learning session. Furthermore, the order of edit where segmentation occurs was varied, ranging from the 5th, 15th, and 25th edit. Similarly, to the previous experiment, a predictive model was trained to process the feature set as input and differentiate the correctness of the student's final solution to the problem as output. This experimental manipulation however aims to

ascertain the ability of the model in terms of predicting the final correctness of student-produced programs earlier rather than later during the learning session. Table 3 illustrates each of these properties for the segmentation window.

Table 3. Window sizes used for 3-gram semantic encoding and order of edit in each windowed segment

Step	Window Size	
	5 Edits	10 Edits
5th		
15th		
25th		

Note. While larger window sizes results in concatenating a larger amount of edits for the purposes of encoding contiguous sequences of meaningful units (i.e., a total of 10 edits), smaller window sizes capture lesser amounts of edits (i.e., a total of 5 edits). An increasing order of edits can be used to determine where segmentation occurs within a sequence, ranging from the 5th, 15th, and 25th edit in the sequence.

In this experiment, the 3-gram semantic encoding approach was retained from the previous experiment and applied to extract features while varying the window size and step used for segmentation of the sequence of edits. Programming trajectories that included less than a total amount of edits covered by the segmentation window were removed from further analysis to control for unequal lengths. Furthermore, the string of text characters that characterize each program state across the set of 5 edits were concatenated as a single example for the purposes of feature extraction.

Table 4. Goodness of fit metric estimates on resampling by type of segmentation window size and step

<i>Metric</i>	<i>5 Edits</i>			<i>10 Edits</i>		
	<i>5th</i>	<i>15th</i>	<i>25th</i>	<i>5th</i>	<i>15th</i>	<i>25th</i>
Recall	.57	.55	.44	.63*	.59*	.56*
F1	.52	.52	.43	.56*	.53*	.49*
Lift	1.10*	1.11	1.13*	1.08	1.10	1.05

Note. 200 examples per paired samples in each condition, controlling for seed of the 5-fold stratified cross-validation ($n = 200$) as a confounding factor. *Significant paired-sample two-sided t-test with bonferroni correction, $p < .005$.

The goodness of fit metric estimates obtained from the resampling procedure by type of segmentation window size and step is given in Table 4. Paired-sample t-test comparisons were conducted to ascertain effects using Bonferroni correction to control family wise error rate. As can be seen from the table, the larger window size that included 10 contiguous edits that were concatenated for the purposes of feature extraction led to better fitting models than the smaller window size under two conditions. First, there is a trend in the results that suggest that the model best fits the positive class when segmentation occurs earlier in a sequence of edits, as both the recall and F1 metric scores decrease inversely to the order of edits. The exam of the scores suggest that best performing models are obtained overall when segmentation occurs on the 5th edit, resulting in a medium effect for recall ($t(199) = 8.34$, $p < .005$, $d = 0.59$) and F1 scores ($t(199) = 7.67$, $p < .005$, $d = 0.54$). Second, the locus of the effect is limited to the class of example detected by the model, as the lift metric is sensitive to positive examples, which are more accurately detected when a smaller segmentation window is used. The effect

depends however on the step or the order of the edit that determines the segmentation, as a medium effect is observed when segmentation occurs at the 25th edit ($t(199) = -8.09$, $p < .005$, $d = 0.57$) and small effect for the 5th edit in a sequence ($t(199) = -3.30$, $p < .005$, $d = 0.23$).

CHAPTER 4 DISCUSSION

4.1 MODEL EVALUATION

The generalizability of the resulting early prediction model was ascertained using a leave-one-out cross validation procedure, where a single learning session is assigned to the test set and the model is trained on the remaining learning sessions. Based on the findings obtained from the prior experiment, the early prediction model segments the sequence of edits that occurred during the learning session at the 5th edit, including a total of 10 edits that are concatenated for the purposes of extracting contiguous sequences of a maximum of 3 semantic units. Table 5 shows the resulting confusion matrix for the resulting model after tuning the depth of partitions made by the decision tree. The exam of the recall metric suggests that the model is able to accurately detect 4 out of 5 incorrect solutions produced by students at the last execution during a learning session (recall = 83.41%). The early prediction model however is not precise, as 3 out of 5 solutions that are predicted as incorrect, were in fact later found to be correctly solved by the students (precision = 58.09%). These findings suggest that students who need assistance are likely to receive it as a result of the early prediction made by the model. However, those that do not require any form of assistance may be presented with instructional content that is not necessary to improve learning and performance.

Table 5. Confusion matrix for the early prediction of student-produced solution correctness at the last execution

Predicted	Observed	
	Incorrect	Correct
Incorrect	176	127
Correct	35	106

Appendix A provides an overview of the branching operations based on the comparison of the total occurrences of different continuous semantic units within the segmented edit sequence during the learning session. The exam of partitions confirms that the decision tree covers aspects of student-produced solutions that differentiate those that are correct from others that are incorrect. For instance, the first or root node in the tree relies on the feature “R2_{_R3”, which is in fact found in the inner loop of the correct solution to this problem. In student-produced solutions where this nested block occurs at least twice (i.e., threshold for R2_{_R3 is greater than 1.5) in the windowed segment of 10 edits, starting at the 5th edit, the likelihood that students will obtain the final correct solution at the last execution is increased. This is most evident in the following decision path to prediction:

Decision Tree Path to Early Prediction of “Correct”

R2_{_R3 > 1.500

| R2_{_f ≤ 0.500

| | R3 ≤ 10.500

| | | f_} ≤ 13.500

| | | | nan_ R2_{} ≤ 0.500: Correct [Incorrect = 7, Correct = 69]

Final Correct Solution

R2{R2{R3{f}R2{1}}f}f

The forward command embedded within the repeat loop (i.e., “R2_{f}”) is not featured within the correct solution, but rather should be replaced with a command to move left. Furthermore, the repeat loop set with a parameter value of 3 repetitions (“R3”) is found within the inner loop of the correct final solution, as well as the forward statement (“f_”). Finally, a value of “nan” denotes a solution where no blocks were included, indicating perhaps that students made a new attempt to solve the problem by deleting their current solution and adding a repeat loop, suggesting that they might be facing difficulties (i.e., “nan_R2_{f}”). A total of 7 learning sessions were incorrectly classified as more likely to be correct at the last execution or attempt to solve the problem, while 69 sessions were accurately detected earlier during the learning session as leading to the correct solution.

4.2 LIMITATION

As shown in the results, the locus of the effect is for encoding at a broader level of granularity to capture series of commands and statements, while the benefits are most evident for the detection of final solutions that are incorrect. Some of the issues emerging from this finding relate specifically to the practical implications for instruction, as model deployment is hindered by underfitting examples with the negative class or correct final solutions. One likely explanation for underfitting examples labelled as correct final solutions at the last execution is the labelling mechanism used in this study. For instance, a fixed size window of length 5 and 10 was used for data preparation, which led to loss of the data when the size of the segmented window was smaller than the threshold value that was set. Sometimes the valuable information could be lost which might lead the model to

misclassify and mis predict. The data processing can be improved such that it includes the first, last and median value of every task session. The above procedure might cause the dataset to be imbalanced, but it will decrease the noise in the dataset.

Another related issue pertaining to the labelling mechanism is the omission of intermediate executions that may occur prior to the last execution during the learning session. For instance, the original dataset might contain 5 contiguous executions, which are likely incorrect, before the student was able to solve the problems by making edits to their solution. The data labelling mechanism fails to capture intermediate edits, and rather focuses solely on whether the final solution was correct or not. The above limitation can be overcome by concatenating the solutions that are false of a particular task session together and similarly for the true solutions for more fine-grained predictions based on time series of edits with non-equal lengths.

4.3 CONCLUSION

Thus, these results provide support for the claim that semantic encoding of textual and compact representations for solution states in block-based programming environments enable models to better predict early during the learning session the correctness of final solutions. The first experiment showed that incorporating semantic encoding with tri-grams led to increased recall and F1 scores. A follow-up experiment was conducted where 10 contiguous edits to define the segmentation window size lead to better recall and F1 scores as compared to 5 contiguous edits. The benefits were most evident when segmentation occurs earlier rather than later during the learning session. Finally, the

resulting model is most able to recognize incorrect solutions at the last execution, but the task of detecting correct final solutions accurately remains a challenge. The implications from an instructional perspective is that students who do not require assistance are nonetheless more likely to receive such extraneous information during a learning session were the model to be deployed for the purposes of personalizing instruction.

Further work is required to ascertain methodological approaches to address this issue as well as to determine the generalizability of this approach to early detection of difficulties across programming tasks and skill components in the context of the block-based learning environment. The other limitation is the algorithm usage for the model training purposes in which other algorithm apart from the decision tree can be implemented and performance could be evaluated. The model suffers from a low precision which might cause problem for classifying correct solution but might work good for classifying the incorrect solution. Also, it is not taken into account at what point of time starting from the task completion session should the suggestions be provided to the students. Thus, the time metric is needed to be taken on account for partial solution suggestions.

BIBLIOGRAPHY

- [1] Anon. 2020a. Computational thinking. (October 2020). Retrieved November 19, 2020 from https://en.wikipedia.org/wiki/Computational_thinking
- [2] Anon. 2020b What is computational thinking? - Introduction to computational thinking - KS3 Computer Science Revision - BBC Bitesize. Retrieved November 19, 2020 from <https://www.bbc.co.uk/bitesize/guides/zp92mp3/revision/>
- [3] Seymour Papert. 1980. Mindstorms: children, computers, and powerful ideas. Basic Books, Inc., USA.
- [4] Jeannette Wing. 2006. Computational Thinking. Communications of the ACM. 49. 33-35. 10.1145/1118178.1118215.
- [5] National Research Council. 2010. Report of a Workshop on the Scope and Nature of Computational Thinking. Washington, DC: The National Academies Press. <https://doi.org/10.17226/12840>.
- [6] National Research Council. 2011. Report of a Workshop on the Pedagogical Aspects of Computational Thinking. Washington, DC: The National Academies Press. <https://doi.org/10.17226/13170>.
- [7] Carnegie Mellon University. 2020. Retrieved November 25, 2020 from <https://www.cmu.edu/teaching/assessment/basics/formative-summative.html>
- [8] R. S. Baker. 2014. Educational Data Mining: An Advance for Intelligent Systems in Education. In IEEE Intelligent Systems, 29, 3, 78-82. DOI: 10.1109/MIS.2014.42.
- [9] Cristóbal Romero, Sebastian Ventura. 2010. Educational Data Mining: A Review of the State of the Art. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions. 40, 601 - 618. DOI: 10.1109/TSMCC.2010.2053532.
- [10] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. ITiCSE-WGR 2015. 41-63.
- [11] K.R. Koedinger, R.S.J.d. Baker, K. Cunningham, A. Skogsholm, B. Leber, J. Stamper. 2010. A Data Repository for the EDM community: The PSLC DataShop. In Romero, C., Ventura, S., Pechenizkiy, M., Baker, R.S.J.d. (Eds.) Handbook of Educational Data Mining. Boca Raton, FL: CRC Press.
- [12] T. Effenberger, R. Pelánek. 2018. Towards making block-based programming activities adaptive. In: Proceedings of the Fifth Annual ACM Conference on Learning at Scale. Association for Computing Machinery, London, United Kingdom.

- [13] M. Resnick, B. Silverman, Y. Kafai, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, and J. Silver. 2009. Scratch: Programming for all. *Commun. ACM* 52, 11, 60.
- [14] B. Harvey and J. Mönig. 2010. Bringing “no ceiling” to Scratch: Can one language serve kids and computer scientists? In J. Clayson & I. Kalas (Eds.), *Proceedings of Constructionism 2010 Conference*. 1–10.
- [15] N. Fraser. 2015. Ten things we’ve learned from Blockly. In *Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 49–50.
- [16] V. Donzeau-Gouge, G. Huet, B. Lang, and G. Kahn. 1984. Programming environments based on structured editors: The MENTOR experience. In D. Barstow, H. E. Shrobe, and E. Sandewall (Eds.), *Interactive Programming Environments*. McGraw Hill.
- [17] R. Benjamin Shapiro and M. Ahrens. 2016. Beyond blocks: Syntax and semantics. *Commun. ACM* 59, 5, 39–41.
- [18] D. Weintrop and U. Wilensky. 2015. Using commutative assessments to compare conceptual understanding in blocksbased and text-based programs. In *Proceedings of the 11th Annual International Conference on International Computing Education Research (ICER’15)*. New York: ACM, 101–110.
- [19] Michael Kölling, Neil Brown, & Amjad Altadmri. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE’15)*. New York: ACM, 29–38.
- [20] Michael Kölling, & Fraser McKay. 2016. Heuristic Evaluation for Novice Programming Systems. *ACM Transactions on Computing Education*. 16, 3, 12:1–12:30.
- [21] David Weintrop. & Uri Wilensky. 2015. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*. Association for Computing Machinery, New York, NY, USA, 199–208. DOI: <https://doi.org/10.1145/2771839.2771860>.
- [22] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education (ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 168–172. DOI: <https://doi.org/10.1145/1999747.1999796>.
- [23] D. Weintrop, & U. Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education (TOCE)*, 18, 1 - 25.

- [24] Thomas W. Price, & 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 (ICER '15). Association for Computing Machinery, New York, NY, USA, 91–99. DOI: <https://doi.org/10.1145/2787622.2787712>.
- [25] Kurt VanLehn. 2006. The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education*. 16. 227–265.
- [26] Tomáš Effenberger, Radek Pelánek, & Jaroslav Čechák. 2020. Exploration of the robustness and generalizability of the additive factors model. In Proceedings of the Tenth International Conference on Learning Analytics & Knowledge (LAK '20). Association for Computing Machinery, New York, NY, USA, 472–479. DOI: <https://doi.org/10.1145/3375462.3375491>.
- [27] M. A. Rubio. 2020. Automated Prediction of Novice Programmer Performance Using Programming Trajectories. In *International Conference on Artificial Intelligence in Education*. 268–272. Springer, Cham.
- [28] Tomáš Effenberger, & Radek Pelánek. 2019. Measuring Students' Performance on Programming Tasks. In Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale (L@S '19). Association for Computing Machinery, New York, NY, USA, Article 26, 1–4. DOI: <https://doi.org/10.1145/3330430.3333639>.
- [29] R. Pelánek, T. Effenberger. 2020. Beyond binary correctness: Classification of students' answers in learning systems. *User Model User-Adap Inter.* DOI: <https://doi.org/10.1007/s11257-020-09265-5>
- [30] Tomáš Effenberger. 2019. Blockly Programming Dataset. In 3rd Educational Data Mining in Computer Science Education (CSEDM) Workshop.
- [31] P. Ihanola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, ... D. Toll. 2016. Educational data mining and learning analytics in programming: literature review and case studies. In *ITiCSE-WGP 2015: Proceedings of the 2015 ITiCSE Conference on Working Group Reports* (pp. 41–63). New York, NY: Association for Computing Machinery, Inc. <https://doi.org/10.1145/2858796.2858798>
- [32] B. Akram, H. Azizoltani, W. Min, E. Wiebe, A. Navied, B. Mott, K.E. Boyer, J. Lester. 2020. Automated assessment of computer science competencies from student programs with gaussian process regression. In *Proceedings of the 13th International Conference on Educational Data Mining (EDM 2015)*. 1, 19.

APPENDIX A. EARLY PREDICTION MODEL

```
R2_{_R3 > 1.500
| R2_{_f > 0.500
| | 1_} > 2.500: false {false=8, true=0}
| | 1_} ≤ 2.500
| | | f_} > 5.500
| | | | R3_{_} > 11.500: false {false=2, true=0}
| | | | R3_{_} ≤ 11.500: true {false=0, true=10}
| | | f_} ≤ 5.500: false {false=4, true=1}
| R2_{_f ≤ 0.500
| | R3 > 10.500
| | | R2 > 8.500: true {false=1, true=5}
| | | R2 ≤ 8.500: false {false=7, true=1}
| | R3 ≤ 10.500
| | | f_} > 13.500: false {false=2, true=0}
| | | f_} ≤ 13.500
| | | | nan_ R2_{_} > 0.500
| | | | | nan > 0.500: false {false=3, true=0}
| | | | | nan ≤ 0.500: true {false=0, true=2}
| | | | nan_ R2_{_} ≤ 0.500: true {false=7, true=69}
R2_{_R3 ≤ 1.500
| }_f_} > 0.500: true {false=1, true=11}
| }_f_} ≤ 0.500
| | nan_ R2 > 0.500
| | | } > 4.500
| | | | R3_{_f > 1.500: true {false=2, true=13}
| | | | R3_{_f ≤ 1.500
| | | | | { > 19.500: false {false=5, true=0}
| | | | | { ≤ 19.500: true {false=17, true=20}
| | | | } ≤ 4.500: true {false=0, true=7}
| | nan_ R2 ≤ 0.500
| | | R4_{_f > 7.500: false {false=10, true=0}
| | | R4_{_f ≤ 7.500
| | | | { > 12.500
| | | | | { > 13.500: false {false=96, true=69}
| | | | | { ≤ 13.500: true {false=1, true=12}
| | | | } ≤ 12.500
| | | | | f_ r > 0.500: true {false=0, true=2}
| | | | | f_ r ≤ 0.500: false {false=45, true=11}
```