Learning Curve Analysis for User Performance in Parsons Problems

Yuyang Geng

**Abstract**

Parsons problems have been proved to help improve efficiency in programming learning. Though previous research suggests that it provides equivalent learning gain as code writing problems, less attention has been paid to finding the difficulties that students have when solving Parsons problems.

In this research, we use Datashop to apply learning curve analysis on the data of students’ submission attempts, based on knowledge components generated from the Abstract Syntax Tree, and use Learnsphere to analyze students’ solution paths. Our ultimate goal is to find the concepts students struggle with when solving Parsons problems and the factors that cause such difficulties, in order to provide clues for further improvement in programming education efficiency. Our analysis also extends the use of learning curve analysis in programming education area to analysis on Parsons problems, lying a foundation for possible future research on other variations of coding problems like faded Parsons problems.

By comparing with the result of previous learning curve studies on code writing problems, we again validate that Parsons problems can improve student’s learning performance and efficiency by investigating the learning curve of single Knowledge Component. We also found that the students mainly struggle with some non-intuitive or uncommon use of some simple KCs, control flow of the code, and the list operation skills, while they may not confuse with some KCs that are supposed to be difficult because Parsons problems may had provided the hardest parts of some skills, e.g. the range in the for loop.

In future, to more precisely allocate the difficulties students has in Parsons problem, we may either let students do think aloud during problem solving process, or make the problem set larger to avoid the bias caused by fixed group of KCs in the code blocks of Parsons problems. Future research may also want to test whether the efficiency-improvement effect of Parsons problems still exist for more advanced programming courses after adjusting the difficulty level of problemsets.

**1 Introduction**

Novice programmers may have hard time start up with writing code. Their unfamiliarity with syntactic and sematic concepts in programming languages frustrates and confuses them. Current programmers mainly learn such skills from spending long time to do large number of practices, making the time to teach a skillful programmer too long. Parsons problems, as suggested in recent research, are now considered a tool since studies have provided evidence that solving Parsons problems can lead to learning gains similar to writing the equivalent code, but in significantly less time (Haynes et al. 2021).

With this premise, we would wonder what concepts students are struggling with over time when doing the Parsons problems, including whether students struggle with similar problems or have different confusions over various concepts.

To answer this question, we would analyze students’ performance using learning curve analysis, a technique from Education Data Mining(EDM) that, according to Rivers et al.(2016), helps track and analyze student learning of knowledge components across multiple problems to determine what and how students are learning and predict if certain skills are well learnt (if the students’ performance matches what theory would predict). Incorporating Parsons problem and EDM’s performance analysis, we expect this research would help provide a guidance for future improvement in programming education and help improve students’ learning performance and efficiency.

**1.1**  **Previous Research**

Many previous research had investigated on students’ difficulties when learning programming skills and concepts. Rivers et al. (2016) uses learning curve analysis to find the concepts that students have difficulty with when writing code. According to this research, we will use the dataset that contains only first-attempts of students’ submission may fit better as a learning curve, while the one containing all-attempts as a reference for more comprehensive information. This research surprisingly reports that the error rate did not decrease as expected through the practices of certain knowledge components because some of them are not well practiced. The research suggests that future research may improve the definition of knowledge component and take students’ design decisions into account.

Helminen et al. continued this research direction, further analyzing students’ programming process and their difficulties using js-parsons to record the users’ interaction with the tool. The environment enables students to get feedback during coding, which causes the occurrence of trial-and-error-like behaviors and abnormal coding logic in the analysis part. The researchers suggested that future research may collect users’ thoughts to avoid speculated sources of coding errors, as well as adding more distractors to intimate real coding situations.

While in the aspects of Parsons problems, Ericson et al. (2022) affirms that Parsons problem can help students learn programming. This research also concludes the limitations of past researches on Parsons problems and suggest that future research should focus more on the use of Parsons problems (1) in advanced programming courses (2) in larger classes (3) for students with different backgrounds (4) with error messages (5) with more participants, etc. In this research, the data was collected from a relatively non-fundamental programming courses with large amount of students enrolled, aiming to also fill in the gap in Parsons problem related research.

**1.2 Research question and expectation**

While learning curve analysis from EDM was applied to research about programming education and Parsons problems are showed to be effective in improving students’ performance, there had been few research about combination of the two to evaluate students’ learning performance and their difficulties in Parsons problems. Based on previous researches, this research would apply knowledge-based learning curve analysis on programming data in this work, in order to investigate on concepts and skills that students may have difficulties with in Parsons problems environment, which is supposed to take more attempts than concepts that are well learnt across students; and within the concepts that students may have different with, if students have different misunderstandings of a concept, the data may have larger deviation from the regression learning curve mode, while if students have similar misunderstandings, they may get a relatively small deviation as well as similar solution paths, indicating that the problem in understanding this concept is caused by the education and we need to refine the way we teach it. Our major research questions are:

o   RQ1. How similar is the concepts that students have when solving Parsons problems compared to those when students solving writing code problems which demonstrated in previous research (Rivers et al. 2016)?

o   RQ2. How similar or different are the process to reach the final solution across the students under Parsons Problems and how is that being influenced by the certain programming skill that the problem contains?

We expect that the confusing concepts would be the same for Parsons problems and writing code problems. For RQ2, we predict that, for problems containing mainly basic concepts like function definition and variable assignments, the problem solving process may be similar across the students and thus the deviation from regression lines will be smaller because they may have learned the concepts/there is little variation in the way to solve such problems; while for more complex problems, like the ones containing loop and if statement, students may have more variation in their problem solving processes. This hypothesis is based on the assumption that students would learn certain concepts before more complex one, while what concepts are more complex will be determined according to the analysis result of the models.

Besides the main research questions, this research is also expected to provide a guide for improvement in future programming education on what concepts are well learnt, what are common difficulties, and which ones are causing confusion across students.

**2**      **Related Work**

**2.1**  **Parsons Problems**

According to Haynes et al.(2021), Parsons problems are “a type of code completion problem that re- quire learners to place mixed-up code blocks in the correct order”.  It has many variations by adding different functions to basic Parsons problems, including feedback functions, extra distractor blocks, fill-in-blank variable operations which are also called faded-Parsons problems, etc. Previous research had proved that Parsons problems can improve students’ programming performance by lowering cognitive load and improving problem solving efficiency (Ericson et al. 2022, Haynes et al. 2021), as well as helping the teachers to identify students’ difficulties.

One significant difference between code writing problems and Parsons problems is that Parsons problems usually have only one correct solution. While this feature makes the data analysis process convenient, it might be considered to influence the difficulty level of the problems as it requires students to read and use code written by others. However, previous studies had evidences that practice with Parsons problems lead to learning gains similar to writing the equivalent code with less time cost (Haynes et al. 2021). As a result, practice with Parsons problems can serve as not only a startup buffer for novice programming students to learn to read, trace, and learn syntax and semantic rules before they really write code, but also a simulation to their future code writing performance when solving similar programming problems.

Parsons problems in this research would mix distractor blocks with correct code blocks to increase difficulty level. According to David et al.(2023), the distractors can also be used to allocate the difficulties that students have by including Parsons problems in formative assessments. Also, Previous research by David et al.(2023) shows that though students spend more time on questions with distractors, no significant influence on score, so the distractors added to the problems are expected to more clearly emphasize the problems that students have difficulty with while not influence their final correctness.

As stated in the research of Haynes et al.(2021), adaptation to Parsons problems is provided in the research setting. The two adaptation methods are developed by Ericson et al (2018) to keep students in Vygotsky’s zone of proximal development, a zone representing the difference between the things learners can by themselves or with support. In the research, the setting of adaptation includes both intra-problem and inter-problem. Intra-problem adaptation is initiated by learner when clicking the “Help” button, resulting removal of a distractor block or combination of two blocks; Inter-problem adaptation initiated by system would modify the difficulty of future problems according to the learners’ performance by adding or removing distractors and pairing distractors with correct blocks. Adaptability of the system helps “maintain desirable difficulties and reduce or eliminate undesirable difficulties”, improving efficiency and engagement in learning (Haynes et al 2021).

For this research, besides RQ1 which is directly related to Parsons problems, Parsons problems could be a better choice than writing code problems to test RQ2, as Parsons problems limits the possible solutions that students can have, as well as limiting the possibilities of problem solving processes of the students since they can only reach the solution by manipulating some fixed code blocks to form a pre-determined solution, making it easier and clearer to find the deviations due to learning difficulties and certain KCs and avoid possible bias due to different coding designs in code writing problems.

A possible drawback for using Parsons Problems would be that the improvement in efficiency is little compared to the code writing problems if the solution of the Parsons problem is not common, and such non-common solution would further influence students’ design in future code writing problems (Haynes et al 2021). To avoid such influence and ensure a fair and comprehensive problem setting, the problem set used in this research are either from past Advanced Placement (AP) Com- puter Science (CS) A exams or from the CodingBat website created by Nick Parlante of Stanford University (Haynes et al. 2021). The concept covered, according to Haynes et al., include strings, lists, ranges, conditionals, loops, dictionaries, and functions, covering the major concepts in introductory computer programming courses.

**2.2 Codespec Programming Tutor**

Codespec Programming Tutor is a self-directed learning environment designed by Haynes et al (2022)to help programming learners to handle basic computer programming skills like code reading, tracing, writing and pattern comprehension and application, to form an adequate mental model of programming concepts, and to provide coding related resources and professional help.

As Figure 1 illustrates, its problem space offers programming problems in multiple forms: pseudocode problem, Parsons problem, Faded Parsons problem (problems that require user to filling blanks in the code blocks), fix code problem, and writing code problem.

The data used by our research is from the users of Codespec solving Parsons problems, recording user activities like starting a new problem, moving a code block, and submission. The left side contains the code blocks that are provided for solving the problem shown on the top part of the screen but in a random order, and the right side shows the answer arranged by the users which the orders and indentations of the code blocks are counted into the log data. When solving a problem, users would drag-and-drop the code blocks from left to right, adjust its indentation, and click on run button for submission and test of the code, counted as one attempt. The bottom part will provide feedback for the code.

This research also aims to provide feedback to the Codespec tool about how to develop models for personalized learning that is fair and effective as an introduction to programming, as suggested in the research of Haynes et al (2022).

Graphical user interface

Description automatically generated

Figure 1. Codespec’s practice area on Parsons problem tab (Haynes et al. 2022)

**2.3 Learning Curve Analysis**

Learning curve analysis is an approach used in education data mining research that estimates the learners’ performance on learning a certain skill under the power law of practice. The law states that with more practice opportunities, the learners would have a lower error rate on a particular skill, and such skills in the context are knowledge components (KC). Statistically, learning curves are fit to students’ performance data, namely, each submission attempts of students for a certain problem in our research, through the Additive Factors Model (AFM) which is a form of logistic regression using students’ prior practice opportunities of certain KCs to predict correctness rate.

In this research, the way to perform learning curve analysis on data will be similar as that in previous research (Rivers et al., 2016).

**3 Methodology**

Among the research questions, we would compare the findings with the results demonstrated in the research of Rivers et al. (2016) to answer RQ1.

To investigate the similarity and differences between students’ learning process (RQ2), we will look at the general deviation of data from the regression curve generated and students’ solution paths. According to Rivers et al., Well-learnt concepts are expected to have a curve with high error rate intercept initially and decrease with more practice of that KC, while no-learning curves may fail to show such a decreasing trend. We may expect the KCs showing a well learnt and already learnt curve to have generally smaller deviation from the regression line, corresponding to similar problem solving process across students, while the no-learning curves have the opposite. If some poorly learnt concepts do have small deviations, we may expect its attempt numbers (opportunity numbers) to be high, indicating that students have similar false conceptions on this programming concept. We may also take an insight of students’ solution path.

**3.1 Material and Data**

The data is a subset from the Runestone dataset of a previous research (Haynes et al. 2021), provided by one of the researchers. The data collection lasts from January 2020 to Faburary 2021 in a post-secondary research institution in the northern Midwest of the United States under the approval of Institutional review board (IRB).  The participants include145 students enrolled in a data-oriented Python programming course which required prior programming experience. The dataset contains the users’ actions on Runestone, a free and interactive eBook platform, and the corresponding time. The users are all anonymized, with numbers as identifications, while some demographic information is collected: 53% of the participants are female and 47% are male; 33% are Asian, 2% are Black, 6% are Hispanic, 46% are White, 4% are Multiracial, and 9% did not specify. The age range is 18-33 with mean of 20 years old and standard deviation of 1.49 years old. 11% are computer science majors, 3% are data science majors, 3% are information science majors, and 83% are others. The average American College Test (ACT) math score was 31 (SD = 3.42) with a range of 1 to 36. The average GPA is 3.721 (SD=0.448).

The subset contains data about user solving 5 simple Parsons problems in Pytho: exp1\_pp1a - has22(H), exp1\_pp3 - diffManMin(E), exp1\_q5\_pp - dictNames(H), Count\_Target\_In\_Range\_Order - countInRange(M), and Total\_Dict\_Values\_PP - dictTotal(M), which E, M, H represents the difficulty level Easy, Medium, and High. Each problem would contain the code blocks that can be rearranged to the correct solution and several distractor code blocks, while each problem has exactly one possible solution and each distractor code block is paired with a correct code block as block 4\_0 and 5\_0, or block 6\_0 and 7\_0 in Fig 1 shows. Users would solve these Parsons problems on Codespec platform. Details about these problems will be provided as supplemental material.

Text

Description automatically generated with low confidence

Fig 1. Each distractor is paired with a correct code block.

Data used to generate the model will contain all submission attempts of the participants to obtain more detailed information about the students’ problem-solving process and record the blocks that have changed from last submission for each attempt. The change of a block can be (1) newly added (2) changed order (3) changed indentation. If a block is removed, it would not be counted as input of next attempt. If the previous submission is empty, then the current attempt will be compared with the attempt before last attempt, since the empty submission is not informative and students may submit by mistake.

We may parse and reformat these data using Python, opening and manipulating it as an excel file for better visualization of the data. The module used for reformatting is openpyxl.

**3.2 Datashop and Learnsphere**

Datashop is an online data analysis service used also by previous research (Rivers et al. 2016) that can perform AFM on datasets and generate learning curves on knowledge component, while Learnsphere is linked to Datashop through the workflow button when analyzing dataset and can create workflow to walk people through the analysis process of the data to reach a final outcome visualization. The dataset imported should have following properties: Anon Student Id, Time, Session Id, Level(), Problem Name, Step Name/Selection/Action, and KC(). We would reformat raw data to fit into this format and use Datashop to analyze of the resulting learning curve models, as well as analyze the students’ solution path through learnsphere.

Among these properties, except the self-explained properties Time and Problem Name, Anon Student Id are identifications for different users; Level is a dataset level which present in a form like Level(Unit), while “Unit” is the dataset level title and the value in the column is the level name; Step Name is the name of a discrete problem-solving step; Selection  is a description of the interface element that the student interact with; and Action is a description of the manipulation applied to the selection. At least one of Step Name, Selection, or Action must have a value for each row, and we choose Step Name here to contain the identifications for the block occurs in the submission that students must have interacted with since the Attempt At Step column requires the value of Step Name, according to the format requirements of Datashop.

In the uploaded dataset, we would examine the individual models of knowledge components. We will mainly look at the shape of the learning curve to check if it shows a well learned pattern and investigate any strange trend; We may also see the standard error of the models shown by the Error Bar.

**3.3 Reformat Code**

A Python module is written to reformat the code from the form downloaded from Codespec to satisfy the Datashop analysis requirements. The files it operates on is a source file and an empty output file, which both are in .xlsx type and modified through the openpyxl module imported but can be exported as csv files.

The original data exported by Codespec, as demonstrated in Figure 1, is in a format that each line indicating an event happen during a certain user solving a certain problem, including starting of a question, moving a code block, submitting an attempt, and so on. Figure 2 illustrates the lines in original data. The code blocks are represented in a form A\_B, while A is the order of the block in the initial state of the problem, while B represents indentation.

A picture containing text, window, appliance

Description automatically generated

Fig 2. original format of raw dataset

Graphical user interface, application

Description automatically generated with medium confidence

Fig 3. indicator for raw dataset

We filter the data to contain only submission attempts and reformat it for Datashop analysis, as shown in Figure 3. While a line in the original format refers to a submission, they each are firstly parsed into multiple lines formatted data which each line refers to a certain code block appears in that submission attempt; later, each code block is further spread to multiple lines that each hold a single KC, which would be talked in the following paragraph.

The data after reformatting would have these properties: Anon Student ID, Time, Student Response Type, Student Response Subtype, Level(default), Step Name, Attempt at Step, Is Last Attempt, Problem Name, Outcome, Input, KC(Tokens) and Session Id. In this case, Anon Student ID represents different users; Time refers to the time that the student submit a certain attempt for a certain problem; Student Response Type is always “submit solution”; Student Response Subtype is always “Parsons problems” since all problems are Parsons problems; Level will always be “programming” indicating the field of research; Step Name is the same as the KC of that line, except when the input is empty or unchanged from last submission it will then be named as problem name + “empty/unchanged in attempt” + Attempt At Step, and that of the KCs that are missing in the last attempt submission will be “missing KC in last attempt”; Attempt At Step indicates the number of attempts submitted for a certain problem by the user; Is Last Attempt is a Boolean which 1 indicates the final attempt submitted by the user for this problem; Problem Name would be one of the five problems; and Session Id identifies the user's session with the tutor which is always 1 in this study as the users are all from the same data collection resource.

The Input here would be a single code block that is contained in the student’s submission attempt, empty for empty/unchanged input, or “missing” +block name for the blocks that are missing in the last attempt submissions. It will record the code blocks that have changed from the last attempt, except that for each user’s first attempts of each problem will record all code blocks. If the input and KC column are empty, it can be (1) the submitted answer is empty, or (2) no blocks are changed/added from last attempt. The Outcome refers to whether that single block is put correctly.

 The last two column, “reference line” and “should be”, are implemented to provide convenience for debugging and future reuse of the code. “Reference line” indicates which input line in the original format of raw data file is the following rows split from, while “should be” indicates what is the corresponding correct code block in the correct answer, or whether the wrong code block is a distractor.

Table

Description automatically generated

Fig 4. reformatted data uploaded to Datashop

The knowledge component (KC) is generated by parsing the solutions provided into Abstract Syntax Trees (ASTs) and identifying the specific constructs used by finding different node types in the tree. The function to generate AST is imported from the ast library in Python and is called through astpretty.pprint(ast.parse(inspect.getsource(has22)).body[0]),which has22 can be replaced by the name of other problems.

To model the learning curve based on single KCs, we create multiple rows for each code block, while each row contains one of the KCs that is learned from placing a correct code block or failed to learn from placing a wrong block or a distractor. In the case of a block at wrong order or with wrong indentation, the KC column contains the KC of the block that should be there, indicating the students fail to learn the KC of the correct blocks. In the case of a distractor, as the distractors are always paired with a correct block, we will use the KC of the corresponding correct code block to fill in the KC column of distractors to indicate what knowledge components the subject failed to learn. And we also manually delete some KCs from both the distractor and its paired code block after checking since they are not really testing that KC, for example, we deleted the If statement KC from code block “if current == target:” and its distractor “if index == target:”, and the Assign KC from “first = p\_dict.get('first', 'Unknown')” and its distractor “first = p\_dict.get('first',None)”. Between the two cases, the latter one has a higher priority.

The code used for reformatting would be attached as supplemental material. To use the code, one should call init(‘source data filename’, ‘target empty sheet’), while both input files should be in .xlsx format. The last three parameters are optional. The third parameter is a string list containing the names of the questions to be filtered out so that one can see data for some out of the total 5 problems, while entering an empty list means setting it as default, meaning to include all 5 problems. The fourth parameter is a Boolean value indicating the switch between the two modes of judging correctness. It’s default true, indicating judging based on the relative order. The two modes are made to get more comprehensive information when analyzing.

The fifth parameter is also a mode controller which controls the Step Name column of the distractors in students’ submissions, which is naturally set to False. If it is set to True, the Step Name of a distractor block in the input will become “distractor A\_B” which A\_B is the distractor block. This mode is set so that when we are investigating strange datapoints on Datashop, we can distinguish if the error rate at that point is caused by a distractor or not to get more information. However, importing data with this parameter set to True will influence the shape and categorization of the learning curves of KC, so data under this mode is imported only to compare with the actual dataset graphs to investigate the sources of error rates.

Table

Description automatically generated

Fig 5. reformatted data when 5th parameter is set to True

Though the current code only contains the 5 problems in this research, it can be easily refined to include more questions by adding new target questions to the answers or by adding an interface.

**3.2.1 Step Correctness and Error Attribution**

A dictionary structure is used to store information about the target questions, using the name of the problem as a key and a list as the value. The first item of the list should be a string list of the code blocks with correct order and correct indentation, representing the solution; the second item should be a string list of the distractor blocks; the third item should be a string list counting the knowledge components in the order of solution, each corresponding to a code block in the solution list; the forth item is a string list of the knowledge components corresponding to the distractors in the second item. The Outcome is determined by comparing the Input block list with the correct answer block list stored in the first item of the value list, as well as identify distractors if there is. The ways to judge correctness would be slightly different in the two modes: based on relative order or absolute order.

In the case that the input is longer than correct answer, which means that the users must add at least one distractor code blocks, we need to insert some fillers into the answer string list to avoid errors. In the relative order mode, the distractor code blocks in the input would be made to compare to some meaningless strings inserted into the indexes corresponding to the indexes of the distractors; in the absolute order mode, if there’s distractor code blocks in the input, same number of meaning less strings would be attached to the end of the answer that the input would compare to. Both ways would result in a temporary answer code block list that have the same number of blocks as the input.

The relative order mode considers the distractor blocks as something added to the answer causing the blocks that could be correct shifted in order, such that it evaluates the relative order of the blocks that are in the correct solution after canceling out the distractors in the answer with strings; while the absolute order mode consider the distractors as something replaces the correct block that should be at its index such that we include the distractors in comparison while make the extra blocks compared to some meaningless strings. The advantage of the relative order is that it makes sure the meaningful code blocks indicating learning of some KCs are compared and evaluated validly, while the absolute order mode may more truly reflect the amount mistakes in students’ learning process. In the research, we would look at the resulting all-attempt models under both modes to have a more comprehensive view of the data.

As a result, taking order and indentation into consideration and following the standard in previous research (Rivers et al. 2016):

* CORRECT if the outcome from Runestone record in the raw data says correct
* If output data says incorrect, we individually check the blocks contained in the submission:
  + INCORRECT if the block is compared to…
    - In relative order mode (parameter == True): “distractor” string, e.g. “2\_1” corresponds to “distractor 2”, while in this situation 2\_1 is definitely a distractor
    - In absolute order mode (parameter == False): “extra” string, e.g. “7\_1” corresponds to “extra 2”, while in this situation, the solution contains the distractor block 2\_x in the input, but it is before the 7\_1 block, such that the existence of distractor 2\_x changes the orders of all blocks after it and make the position of 7\_1 longer than the length of the correct solution, making it can only compared to the fill-in string (“extra”).
  + INCORRECT if the block is different from/has different indentation with the block on its index in the solution block list, e.g. “2\_1” corresponds to “3\_1”, or “2\_1” corresponds to “2\_2”, such that its corresponding KC are not learned.
  + CORRECT if the block is the same as the block on its index in the solution, “2\_1” corresponds to “2\_1”, such that the corresponding KC is learned.

In our research, we say that all KC opportunities are evaluated in the first attempts, but in following submissions we only look at those which changed after the previous attempt. Then, in the order of priority, the KC outcome is corresponded to the correctness of the input code block:

* If the KC is in the student’s solution but not in the goal solution or with a wrong order or indentation, it is INCORRECT (namely, the KCs of the wrong code blocks are INCORRECT).
* If the KC occurs in the edit between previous and current state (or this is the first attempt), AND is on its correct order and with correct indentation, it is CORRECT.
* If the KC is missing from the student’s solution, and this is the last attempt,  it is considered INCORRECT.
* If the submission is unchanged from last attempt and correct, the KCs are not counted to avoid double counting learning of KC.
* Otherwise, it is skipped for this step.

While this research has a relatively strict standard on whether a KC is learned, since previous research like Rivers et al.(2016) did not take indentation and order into consideration, this research may provide some different insights.

As we have the reformatted data of all submissions, we may use the all-attempt model as a reference for categorization and more comprehensive information when needed. We may focus on the first-attempt of students’ submissions as that is what actually reflects practice opportunities of KCs and is proved to fit learning trend in Rivers et al.(2016). The first-attempt data is a subset of the all-attempt data by setting Attempt At Step = 1 as a filter condition.

**4 Results**

Learning curve models of knowledge components are supposed to start with a relatively high error rate and decrease as practice opportunities increase. In Datashop, there are five categories of individual learning curves: curves that data is not enough for solid analysis (little-data), curves that start and end with low error rates (already-learned), curves that start and end with high error rates without decreasing (no-learning), curves that start and end with high error rates with some learning phenomena (still-learning), and curves that start with high error rate but end with low error rate (good-learning). While we expect to see good-learning curves, indicating students learned the concepts well, other curves are also informative.

Interestingly, when we compare our standard model (first attempt data under mode 1) with others, the result model of first-attempts data and data under mode 2 shows a similarity in shapes of all curves and a few differences in categorization of some KCs like FunctionDef and For. Firstly, we will look at the whole-dataset learning curve of all KCs and all students. It starts with an error rate at about 38.451%. The curve generally shows a good learning trend that start with relatively high error rate and gradually descends. Through there is a sudden increase near the end of the curve, as well as the individual curves of some KCs, it is reasonable since missing KCs in last attempt are counted as incorrect, causing the error rate in the last few attempt submissions increase rapidly compared to previous attempts. The curve ends with an error rate of 15.978%, obviously lower than what it started with.

In the following analysis, we may mainly focus on the relative order, first-attempt models as it fits people’s intuitive judgement of correctness and only first attempts of problems are considered as a practice opportunity, while we may also look at the data in the all-attempt model to get more comprehensive information.

We will then examine the individual KC learning curves within the First-attempt Modified-Step model and found the following categorizations for all AST node KCs according to the classification on Datashop:

* Little-data: Add, And, Assign Operator +=, Attribute, BoolOp, Comparison Operator < Less than, If Statement, Sub, While, append() function, get() function, len() function, max() function, range() function
* Already-learned(Low and Flat): N/A
* No-learning: BinOp, For, FunctionDef, args,
* Still-learning (Still high):Assign, Comparison Operator ==
* Good-learning: Return, Subscript

As models in Little-data category are not representative enough to provide much information about students’ performance in those KCs, we will mainly focus on the other four categories and see what information they might indicate.

**4.1 Good-Learning Curves**

We may first look at the successful learning curves.

The only two learning curves in this category are Return and Subscript. Interestingly, while Return occurs in all five problems included, Subscript only occurs in three out of the five problems. It is also intuitive to see Return being classified in this category as it is usually considered one of the relatively basic programming skills.

The curve of Return starts at an error rate of about 37.736%. The curve shows an obvious and continuous decreasing trend which finally ends with an error rate of about 2.881%. While the curve of Subscript is not that standard. It starts with an error rate of 66.355% and continuously decrease before opportunity 4. However, the error rate increased suddenly after opportunity 4 and ends with about 52.926%. The data shows that at opportunity 4, there are still 10 observations of learning of Subscript KC with about 3 observations are incorrect, resulting in an error rate of 30%, while in opportunity 5, the number of observations decreased to 3 which are all incorrect. The source of error at opportunity 5 is exp\_pp1a and Count\_Target\_In\_Range\_Order. The major source of incorrectness in Subscript KC in exp\_pp1a, according to the error report, is because students confuse between “if nums[i] == 2 and nums[i-1] == 2:” and the distractor “if nums[i] == 2 and nums[i+1] == 2:”, in other words, choosing the paired distractor instead of the correct block. The source of error in Count\_Target\_In\_Range\_Order is more complexed: As this problem includes an if statement nested in a for loop, the source of error includes not only problems about subscript like choosing the wrong index of the list(namely, choosing the paired distractor), but also includes blocks that are influenced by confusion about more advanced problems like the difference between iteration based on item in a list and iteration based on their index. While the error rate of Subscript did not decrease a lot in the first-attempt model, the shape in the all-attempt model do show an obvious and contiguous descending trend. Generally, in the two learning curves, a small proportion of students may still have confusions about the use of these skills in some complex contexts, but it is expected to improve with more practice opportunities provided.

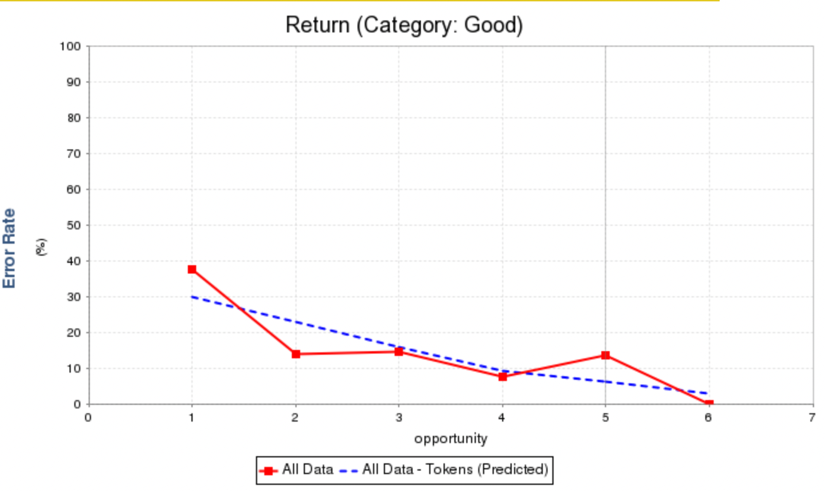


Fig 6. First-attempt modified-steps model of Return

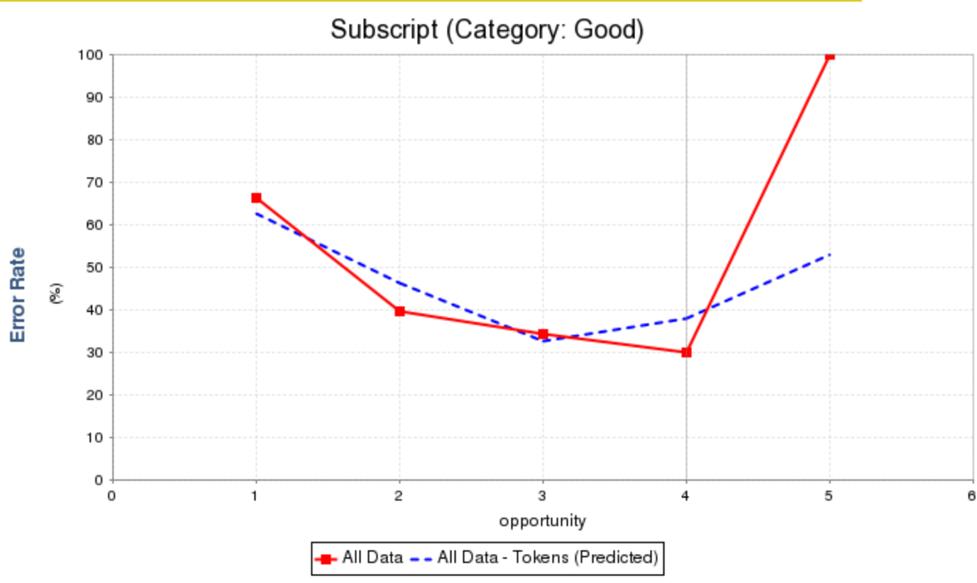


Fig 7. First-attempt modified-steps model of Subscript

**4.3 Still-Learning Curves**

The Still-learning curves, also called Still-high curves on Datashop, are curves seems to trend downwards but fail to reach an error rate that is low enough to confirm that the concept is learned. Among the two KCs in this category, Assign and Comparison Operator == Equal (Equal), it is more reasonable to see Equal be categorized as still learning as Assign is a basic programming skill which occurs in all of the problems and is expected to gain enough practice opportunities.

The general shape of the Assign curve shows a descending trend in the first half, starting with an error rate of 59.813%. The error rate increases from opportunity 5, reaches the peak at opportunity 8 with value of 62.551%, and finally ends with 15.978% at opportunity 9. The error report shows that the source of error mainly include: (1) wrong indentation (2) mistake in ordering of blocks (confusion with control flow), and (3) selecting paired distractors instead of the correct block in problems. While the first two kinds are not related to understanding of specific skills, it seems that the mistakes in the third category are also not closely related to the Assign KC in our data: they include confusions in the value that should be assigned to a variable (e.g. using the distractor “i = 0” instead of “i = 1” is the biggest source of incorrectness of Assign skill in exp1\_pp1a , or the inverse in Count\_Target\_In\_Range\_Order), missing blocks like “count = count + 1” in Count\_Target\_In\_Range\_Order (which is related to misunderstanding in For loop KC), or is due to the confusion on other KCs. The last case is so common that raises the error rate of Assign KC because Assign KC usually occurs with other KCs and is reasonable to be confusing as there are fewer practice opportunities of those KCs. For example, when concatenating a string, use “name = first + last” instead of “name = first + " " + last” in exp1\_q5\_pp, this block is actually also related to the Add KC, and some skill of function call KCs are needed for using first = p\_dict.get('first',None) instead of first = p\_dict.get('first', 'Unknown')  when searching a dictionary in exp\_q5\_pp).  It is reasonable to say that the increment of error rate of Assign KC in the second half is mainly caused by its commonality which makes it usually occur with some more advanced KCs, instead of itself being a confusion for students.

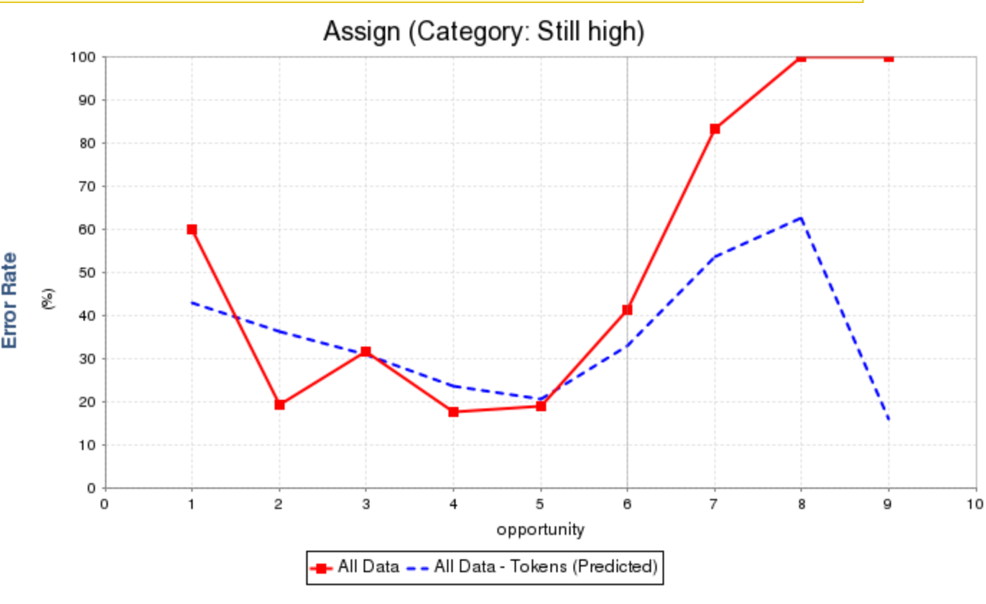


Fig 8. First-attempt modified-steps model of Assign

The shape of Equal, however, is somewhat different from that of Assign KC. It starts with a high error rate of 63.259% and ends with a higher error rate of 91.035%, with a trivial descending trend in the first half of the curve. Although the shapes are different, investigation of Equal curve has very similar reasons behind those errors and supports the idea that the error rate of a KC may increase not necessarily because students are confused with it as we mentioned in the analysis of Assign curve, but because students are struggling with some other KC that usually occur with it. For example, the error report shows that in the problem “Count\_Target\_In\_Range\_Order”, the error rate of Equal KC is influenced a lot: about 29.76% of all observations of Equal KC are incorrect due to some students choose the paired distractor “if current == target:” instead of the correct block “if index == target”, while this mistake actually is not only related to the learning of the equal sign == but also related to confusion between list item and list index. One way to eliminate this influence from the learning curves is to collect data using a larger problem set such that there will be larger variation in the grouping of KCs, or letting the students think aloud as they solve the problems, such that we can get a clearer idea about exactly what concepts are confusing the students and emphasize the explanation of those concepts in future education.

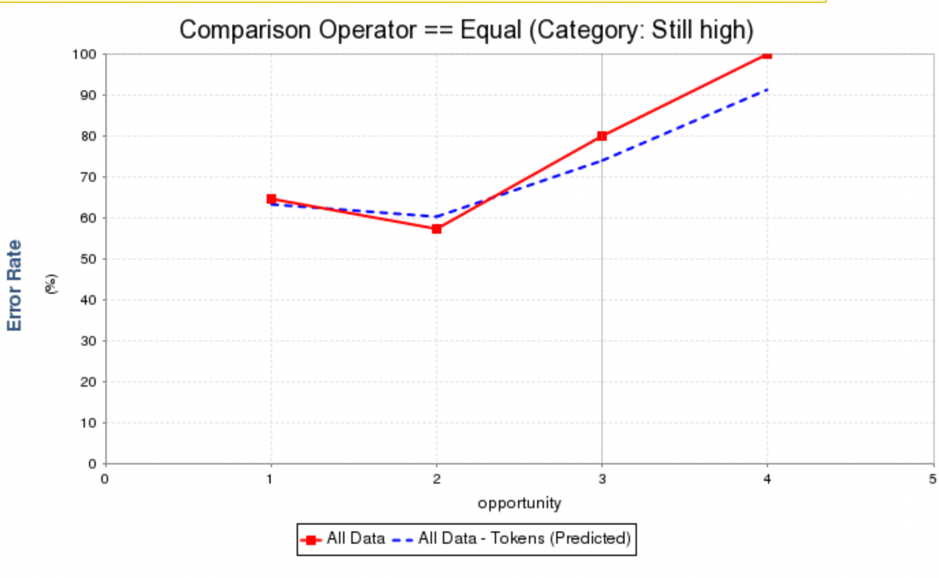


Fig 9. First-attempt modified-steps model of Equal

**4.4 No learning Curves**

No-learning means that the curve of Add KC starts and ends with a high error rate while does not show a downward trending. Among these graphs, the shape of BinOp graph is similar to that of For KC, while the shape of FunctionDef graph and graph of args are very similar.

As we can see in the graph of BinOp KC, the error rate starts from an error rate of about 24.402% and ends with an error rate of 15.978%. It has a peak at opportunity 5 with error rate of 83.927% but then dropped to 15.979%, probably because there’s only one observation after opportunity 4. We investigated the causes of the increment in error rate from opportunity 3 and found students struggling with the following problems: (1) in Count\_Target\_In\_Range\_Order problem,  choosing the distractor “for index in range(start, end):” instead of the correct block “for index in range(start, end+1):”, or using the distractor “count++” instead of “count = count + 1”;(2) choosing the distractor “name = first + last” instead of the correct block “name = first + “”+ last” in exp\_5\_pp problem, and (3) adding the variable to the list before modifying the variable, in other words, switch the position of name = first + " " + last (or its paired distractor) with that of “name\_list.append(name)” in exp\_5\_pp. From these causes, we can see that other KCs that are used together with BinOp has a large influence on its error rate, such as the definition of range () function call and the concatenation of strings.

We also notice that one of the problem student struggles with is using the distractor “count++” instead of “count = count + 1”.  in Count\_Target\_In\_Range\_Order. This is a reasonable confusion since the difference between these two code blocks are not likely to be taught in an introductory programming class and should be considered in future problem design.

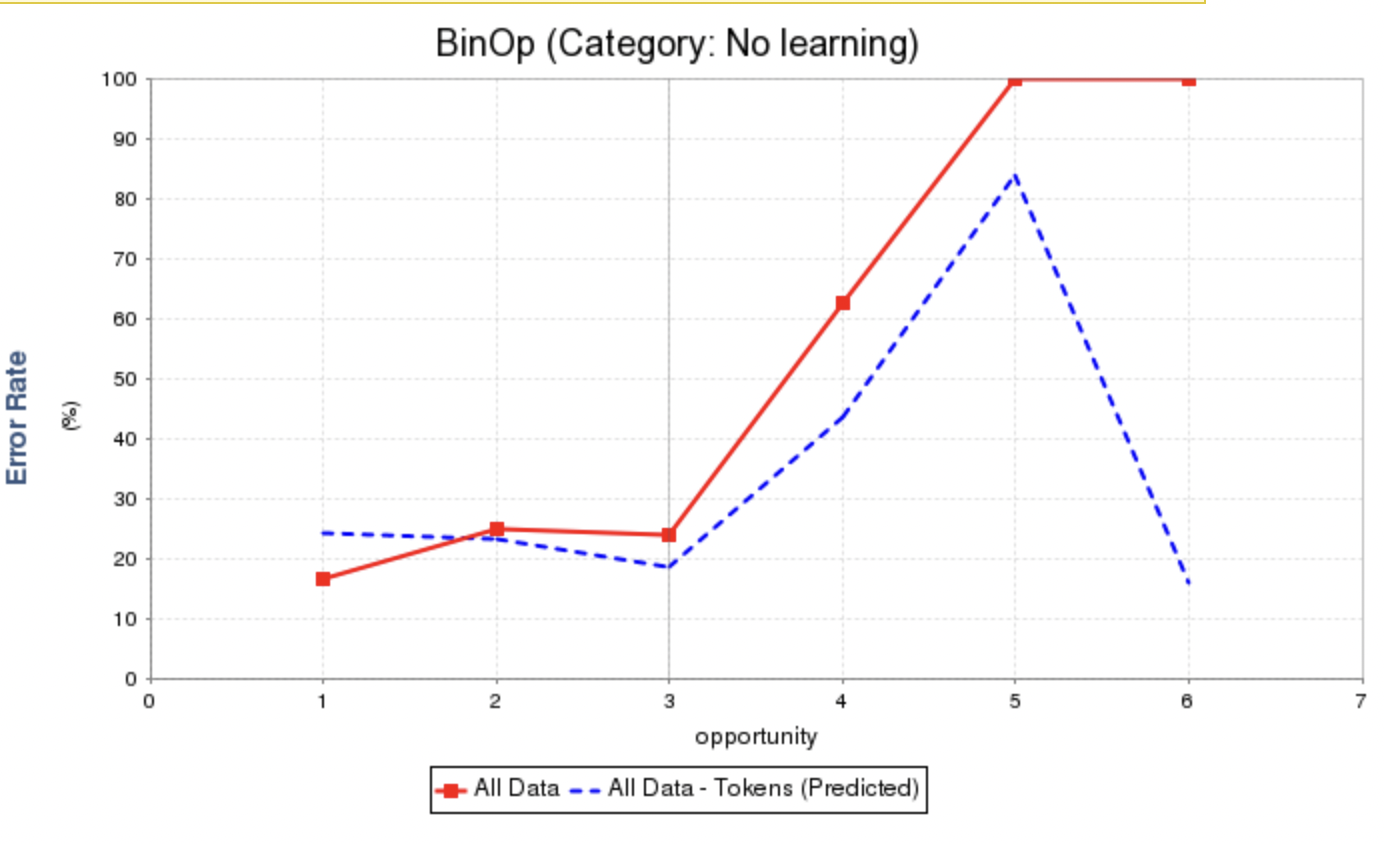


Fig 10. First-attempt modified-steps model of BinOp

The curve of For KC starts with an error rate of 25.743% and ends with 85.65%. It is almost flat before opportunity 3 with more than 74 observations and starts increasing after that with fewer than 3 observations. This means that most students finished learning For KC with a few practice opportunities and a relatively low error rate. We investigated the increase of error rate at opportunity 4 and found it is mainly due to (1) same reason as the reason (1) of BinOp, choosing the distractor block which has the wrong range in Count\_Target\_In\_Range\_Order, or (2) switch position of the For loop block and a loop body block or initialization block in exp1\_q5\_pp. This might be evidence that students are confused with the advanced concepts related to For KC, like the control flow, instead of the basic use of it.

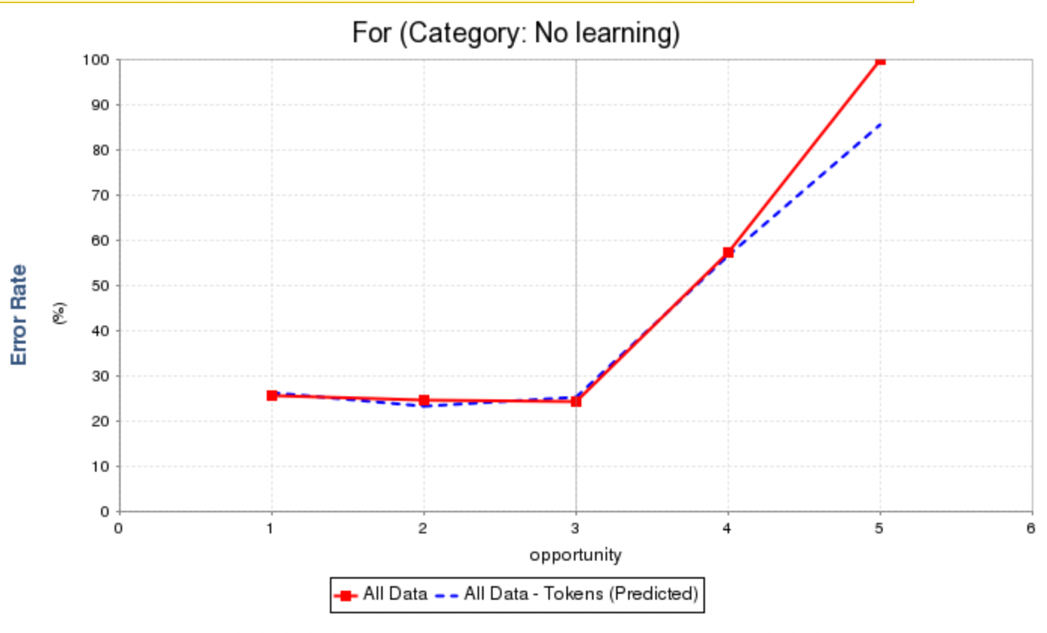


Fig 11. First-attempt modified-steps model of For

For FunctionDef and args, they both occurs in all of the five problems and usually occurs together in one code block, which makes their shape similar. The shape of these two graphs is actually low and flat, and they are classified as Already-learned in the all-attempt model. As a result, we may assume that students can master these skills with a few more practice opportunities.

The curve of FunctionDef starts at an error rate at about 3.738% and ends with 21.52%. While the actual datapoint at the end of the curve is 0%, there’s an uptick at opportunity 8 of 23.077% which lifts up the error rate. We investigated the uptick and found that was because a student changed the function definition code from “def total\_values(dict):” to its paired distractor “def total\_values():” in TotalDictValue problem, which means that the student do still have confusion about the use of parameter in function definition while the fixed code blocks of Parsons problems somewhat lower the difficulty level of these KCs; otherwise, in code writing problem, if the student add the parameter “dict” in function definition, it may probably not be a problem for the code to run, but may cause more confusion in writing the code body.

Since the two KCs usually occurs together, it’s reasonable to see that the args learning curve is similar with the FunctionDef curve. It starts at an error rate of 6.542% and have the uptick at opportunity 7. It has two strange peak at opportunity 6 and 8 which lifted the error rate data of the learning curve at opportunity 7,which, except the same error source mentioned above for FunctionDef, is mainly due to missing KC at last attempt, including (1)missing the step that add the new item into a list “name\_list.append(name)” in exp1\_q5\_pp, (2) missing “for index in range(start, end+1):” in Count\_Target\_In\_Range\_Order. These two reasons, as we mentioned in previous error rate compositions, shows that the error is probably not related to confusion about args KC, but instead confusion about list iteration and operation skills. The former one also shows that as args KC usually occurs along with some function call KCs and these KCs are all categorized into Little-data, this would in some degrees influence the error rate of args KC. However, though students could easily get wrong on those function calls, they may have little influence on the learning performance of args KC as the number of practice opportunities on the function call KCs are relatively small. But we should still pay attention to the error attribution effect in future research with larger problem sets which args may occur with these more complexed programming skills.



Fig 12. First-attempt modified-steps model of FunctionDef

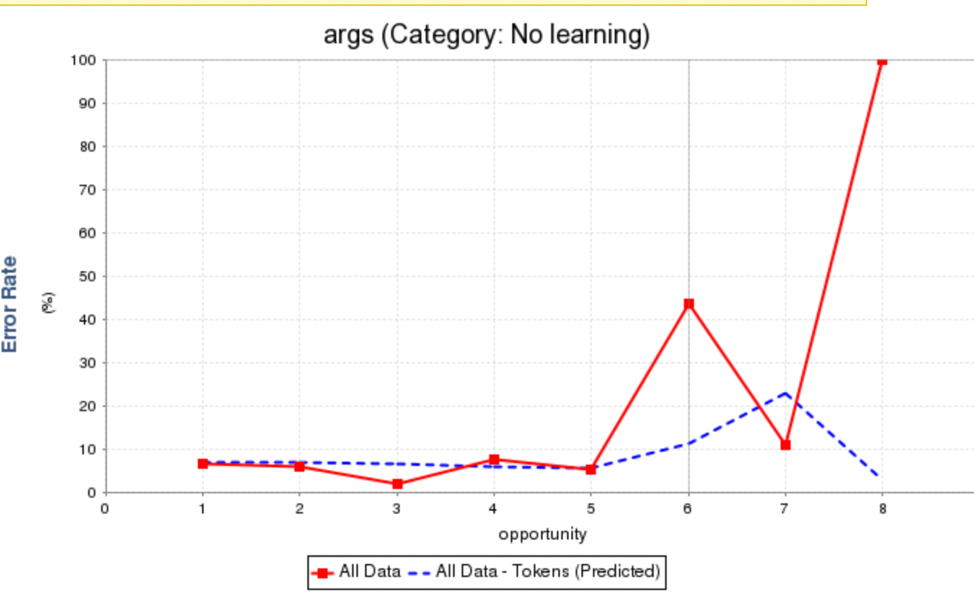


Fig 13. First-attempt modified-steps model of args

**4.5 Little data curves**

There are some curves that has enough data points in all-attempt curves but not in first attempt curve. We may also look at the curve of Add and Assign Operator += briefly and see what can be investigated.

The shape of Add KC graph is similar to that of BinOp KC. If we look at the problems, we will find that in the four code blocks that has BinOp KC, three of them also have Add as its KC. As we can see in the graph of Add KC, the error rate starts from an error rate of about 28.871% and ends with an error rate of 15.848%. The error rate increased from opportunity 2 due to missing KC in last attempt and and reaches a peak of 88.926% at opportunity 5 and 6 that only has a single observation. We investigated the causes of the increment in error rate from opportunity 3 and found students struggling with the following problems: (1) in Count\_Target\_In\_Range\_Order problem,  missing the block “count = count + 1” or “for index in range(start, end+1)”; (2) choosing the distractor “name = first + last” instead of the correct block “name = first + “”+ last” is the second cause of incorrectness of Add KC in exp\_5\_pp problem. From these causes, we can see that the concept that students actually confuse with is not the concept Add, but other KCs that are used together with Add, such as the definition of range () function call and the concatenation of strings. The investigation of the Add KC also exposes students’ confusion about list skills.

Assignment Operator += may be most special among these KCs as it is the only one that occurs in two of the five problem, which is fewer than half of all problems, while other problems all occur in at least three of the problems. But it does show a relatively smooth downward learning curve and have enough observations for each opportunity, from 107 observations in opportunity 1 to 80 observations in opportunity 2, and then 17 student struggle with it and got to opportunity 3, and finally successfully result in a 6.489% error rate with a single observation of 0% error rate. It seems that students can quickly learn this concept through relatively small number of practices. As a sub-concept of the Add concept when it used mathematically, this help proves that the part of concepts students have difficulty with is not the arithmetical Add KC, but the more advanced use of Add sign.

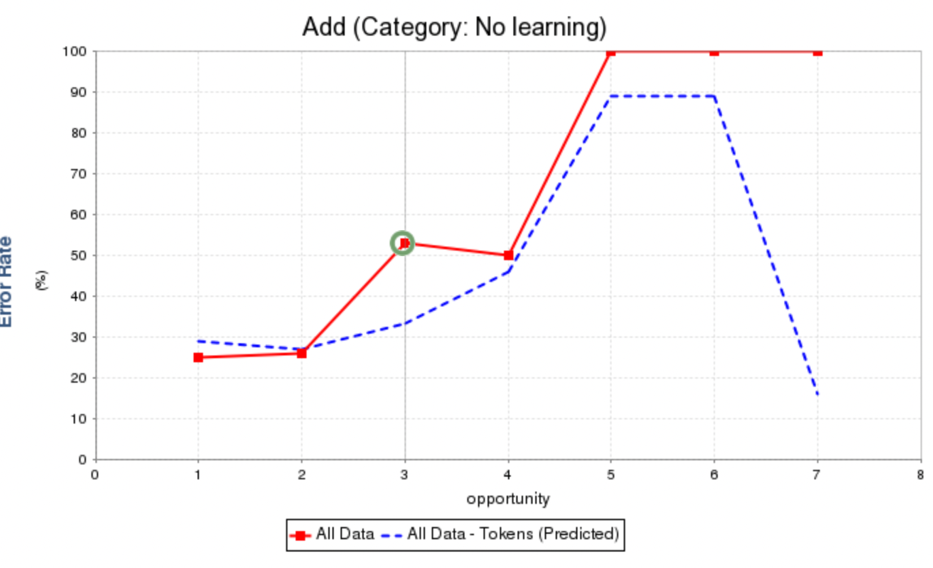


Fig 14. All-attempt modified-steps model of Add

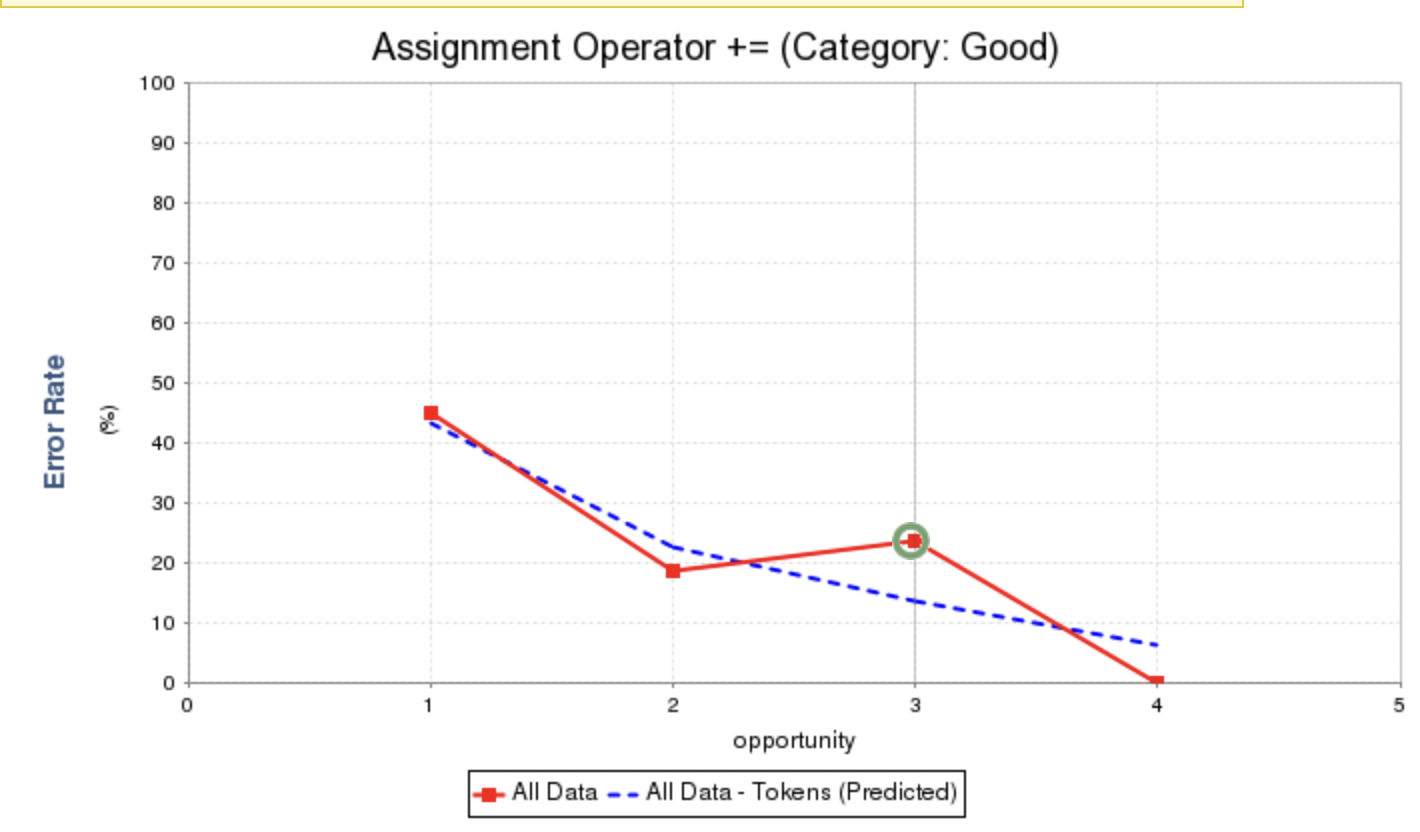


Fig 15. All-attempt modified-steps model of Assignment Operator +=

**5 Discussion**

From the analysis of the KC curves, we know that students, mainly struggle with the advanced skills about some KCs instead of their basic use, e.g. BinOp and Add for concatenating strings, the use of For KC in control flows, and list related operations. This notify us that the error attribution of the different uses of the same code in different contexts may represent different KC or skills, for example, when the plus sign is used in mathematic operations, its KC should be distinguished from that used in string concatenating. While we look at the all-attempt model, we may notice that the FunctionDef KC is classified as Already-learnt, For is classified as Still-learning, and Assign is classified as already learned, indicating that students would be able to proceed and master these KCs with more practice opportunities provided. Another interesting observation is that most reasons behind the error rates are distractors and indentations instead of confusion about the order of the blocks even though we have a complexed problem with nested loops and if conditions. We doubt that the error rate of control flow related KCs is influenced since the code block in Parsons problems provides the most confusing part of using for loop, the expression 2 part of “for <expression 1> in <expression 2>”, to the students. But it could be either positive or negative influence: students may be either benefited from this because they can understand the control flow more easily than designing one by themselves, or they can be confused because the provided control flow does not fit with their intuition. One example is the error attribution of For KC in exp1\_q5\_pp and that in Count\_Target\_In\_Range\_Order. While the latter one has more complexed control flow, the error of For KC does not include mistakes in order of blocks, but the order problem does occur in exp1\_q5\_pp problem that has a simpler for loop structure.

Then, we compare the findings with the results demonstrated in the research of Rivers et al. (2016) to answer RQ1. Previous research of Rivers et al.(2016) performed this analysis with code writing problems and result in the following categorization, with the KCs with too little data excluded since they are hard to analysis:

* Already-learned: If, Module
* No-learning: <, >, Add, And, Assign, Attribute, Binary Operation, Boolean Operation, For, Index, Integer Divide, Load, Modulo, Multiply, Name, Number, Parameter, Power, Return, Store, String, Subscript
* Still-learning: ==, Call
* Good-learning: Arguments, Compare, Function Definition

Excluding the KCs that only one side has, the overlapped items are: BinOp, For, Equal(==). However, as our analysis states, the error rate of == seems to be influenced by some relatively advanced KCs like If statement or range() function call. While one point we can see from the result of Rivers et al. is that larger problem set does solve the worry that two KCs that occurs together in a code block would influence each other’s error rate. Though the actual situation may be different since Parsons problems are code blocks that may always group certain KCs together and are less flexible than the measurement of KC correctness in code writing problems, this is worth testing in future.

The ones that are categorized differently, in the format of our categorization/categorization of previous study, are (1) No-learning/Good-learning: FunctionDef, args; (2) Still-learning/No-learning: Assign; (3) Good-Learning /No-learning: Return, Subscript. For the Arguments (args) and FunctionDef KC, their shapes are actually low and flat that is very close to the shape of Already-learned curves, so we can also consider them as curves that are similar to that in the study of Rivers et al. Besides (1), all other differently categorized items show a better learning trend under practice of Parsons problems than that under practices with code writing problems. We may thus conclude that in general, for the overlapping KCs, students have a better learning performance when solving Parsons problems than code writing problems.

**5.1 Limitations**

There are some limitations of the research. Firstly, though the concepts (KCs) included in the problems are all basic Python programming skills, the learners are students who have prior programming experiences, which means that most students can possibly learnt these concepts and skills before they start the problem set. As a result, there could be bias in catching difficulties during the learning processes of those already-learnt KCs, probably including fewer KCs that students had difficulty with during learning than it should really be.

Secondly, as River et al. (2016) states, learning curve analysis and AFM are not initially designed to be used like this. The way we fetch KCs may also deviate from the standard. However, since our main purpose is to do comparison of the models generated through same steps and standard, this problem could have minimal negative influence on our final findings.

Thirdly, the knowledge component is not defined precisely: they are defined as a group to correspond to a certain code block. Specifically, a student may choose a distractor that correctly use if statement but wrong with the number assigned to the variable, but in our definition neither the KC of if statement nor the KC of assignment would consider to be learned in this submission attempt; and an INCORRECT due to false indentation may indicate that one had learned the KCs but just not the control flow. Fortunately, for the situations that students choose a paired distractor instead of the correct code would probably not cause severe bias on our research result, since previous research by David et al.(2023) shows that though students spend more time on questions with distractors, no significant influence on score, so distractor won’t become a major difficulty that students have problem with. While this problem may not be that important in the introductory level of programming courses, as these bonded KCs do exist together for most of the time and are reasonable to be considered as a whole, it may have bigger influence on the accuracy of data and modeled curves in more advanced programming courses as there are more variations in combinations of KCs.

Also, assignment of unlearned KCs may need to be more detailed for distractors. The distractors may not only replace the paired correct block, but also be placed in a wrong order, so the KC of the distractors (KC that is not learned) should contain both KCs of its paired block but also the block that should on its position, or either, according to specific situation. In the current data format, we only implement either KCs of the paired block or KCs of the block that is replaced by the distractor and place higher priority for the latter situation. This can have some influence on the descending rate of learning curves.

**5.2 Future Work**

As stated in the third limitation of the current research, we may want to include a think aloud protocol when collecting future data such that we can know more detailed about which KCs are learnt and which are not. For example, in the case that the user placed a for loop code block “for x in range(0, 10)” instead of the correct while loop code block “while x <= 10”, we can get information about whether the student does know what the code is aiming to do (create a loop) but just uses the wrong code block, or the students is completely going on a wrong way. In the former case, we can annotate the other KCs of the code block as learnt and annotate only the while loop KC as not learnt, while in the latter case, all KCs are unlearned.

Besides, future research session may also collect data in a larger scale course or in a more advanced programming course such that the problem set can include more problems with various KCs.  As we described before, more variation in KCs and types of problems tested is important to detect what exact concepts students are struggling with, since in Parsons problems, KCs are grouped together in code blocks and we cannot distinguish which KC students are confused with if they get the whole code block wrong. This situation will be much improved when the KCs occur in various different problems and are grouped with various KCs because then we will know which KC students are bad at if they always get the related code blocks wrong.

While making large problem set may have worries in the standards of a good Parsons problem, using Faded Parsons problems can also provide more flexibility in measuring the correctness of KCs in students’ submissions and thus more precisely measure student’s learning behavior than that of Parsons problems. This could probably provide more comprehensive details for future programming education studies.

Also, Future research may also focus on analysis on learning process of more advanced programmers with more difficult Parsons problem sets. Data from more advanced programming courses are needed to investigate further questions like whether the phenomenon is common across current programming education and factors that are causing confusions, e.g. if the students have difficulty in list in Python, would they also confuse with list in Java/C/other languages.

The adjustment of difficulty level can not necessarily be harder learning concepts or including more knowledge components; it can also be other variations of coding problems like faded Parsons problems. The research of Flynn et al.(2023) had uncovered the influence of different fading strategies on the difficulty level of problems and thus influence students’ performance based on the number of attempts and time students need for solving a problem. According to the research, conditional fading led to more difficult problems than variable fading and operator fading, while all fading questions consume more time than standard Parsons problems. By utilizing distractors and fading strategies, it’s possible to gradually improve the difficulty level of the problems to transform them from standard Parsons problem to a form that close to code writing problem, covering the whole learning process for a programmer to understand and master a programming concept.

**6 Conclusion**

 Parsons problems are agreed to be helpful for programming learning with an equivalent learning effect as writing code problems and higher efficiency and lower cognitive load (Haynes et al. 2021). In this research, we find that by practicing with Parsons problems, students are able to learn basic programming skills and concepts more efficiently than practicing with code writing problems, while it is unknown that if this improvement in efficiency continues for learning of more advanced programming skills.

We expect this work can provide as a support for future design of programming courses and research about students’ performance and difficulties.

**Citation**

Barbara J Ericson, James D Foley, and Jochen Rick. (2018). Evaluating the efficiency  
        and effectiveness of adaptive parsons problems. In Proceedings of the 2018 ACM Conference on International Computing Education Research. 60–68.

Carl Christopher Haynes-Magyar and Nathaniel James Haynes-Magyar. (2022).  
           Codespec: A Computer Programming Practice Environment. In Proceedings of [62] the 2022 ACM Conference on International Computing Education Research-Volume  
            2. 32–34.

David H. Smith and Craig Zilles. (2023). Discovering, Autogenerating, and Evaluating Distractors for Python Parsons Problems in CS1. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023). Association for Computing Machinery, New York, NY, USA, 924–930. <https://doi.org/10.1145/3545945.3569801>

Ericson, B. J., Denny, P., Prather, J., Duran, R., Hellas, A., Leinonen, J., ... & Rodger, S. H. (2022). Parsons Problems and Beyond: Systematic Literature Review and Empirical             Study Designs. Proceedings of the 2022 Working Group Reports on Innovation and             Technology in Computer Science Education, 191-234.

Flynn Fromont, Hiruna Jayamanne, and Paul Denny. (2023). Exploring the Difficulty of Faded    Parsons Problems for Programming Education. In Proceedings of the 25th Australasian Computing Education Conference (ACE '23). Association for Computing Machinery,  New York, NY, USA, 113–122. https://doi.org/10.1145/3576123.3576136

Haynes, C., & Ericson, B. (2021, May 1). *Problem-solving efficiency and cognitive load for adaptive Parsons problems vs. writing the equivalent code: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM Conferences. Retrieved  March 22, 2023, from https://dl.acm.org/doi/pdf/10.1145/3411764.3445292

Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L. (2012, September). 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 (pp. 119-126).

Rivers, K., Harpstead, E., & Koedinger, K. R. (2016, September). Learning curve analysis for programming: Which concepts do students struggle with?. In ICER (Vol. 16, pp. 143- 151).