Learning Curve Analysis for Programming: Which Concepts do Students Struggle With?

Kelly Rivers, Erik Harpstead, Ken Koedinger
Human-Computer Interaction Institute, Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15232
{krivers, eharpste, koedinger}@cs.cmu.edu

Abstract
The recent surge in interest in using educational data mining on student written programs has led to discoveries about which compiler errors students encounter while they are learning how to program. However, less attention has been paid to the actual code that students produce. In this paper, we investigate programming data by using learning curve analysis to determine which programming elements students struggle with the most when learning in Python. Our analysis extends the traditional use of learning curve analysis to include less structured data, and also reveals new possibilities for when to teach students new programming concepts. One particular discovery is that while we find evidence of student learning in some cases (for example, in function definitions and comparisons), there are other programming elements which do not demonstrate typical learning. In those cases, we discuss how further changes to the model could affect both demonstrated learning and our understanding of the different concepts that students learn.

Keywords
learning curve analysis; educational data mining; programming syntax; knowledge components

1. Introduction
In recent years there has been growing interest in using large collections of logged programming data to understand how students learn, what they struggle with, and what we can do to improve computer science education. This trend is occurring at the same time as a rise in the use of Educational Data Mining (EDM) [2], a field of study which has developed many useful new approaches for analyzing and interpreting collected student data. However, the majority of research done on programming data has only used metric approaches that cover easily measurable content, such as the compiler errors students encounter and their typical working behaviors [10]. These studies have taught us a great deal about how students write code, but they have mostly examined the output of code, instead of investigating how the code that students write changes over time. It is possible that, by ignoring the written code that students produce, we are missing out on a great deal of useful information.

ICER '16, September 08 - 12, 2016, Melbourne, VIC, Australia.

ACM 978-1-4503-4449-4/16/09…$15.00.
DOI: http://dx.doi.org/10.1145/2960310.2960333

The broader field of EDM investigates how details of student work can be used to determine what and how students are learning. Leveraging theories about the ideal shape of learning [14] and the structure of student knowledge [13], researchers in this space have developed methods for tracking student learning of knowledge components across multiple problems [4, 6], to see if students' ability to solve problem steps associated with particular concepts matches what theory would predict. Discrepancies between theory's predictions and students' behaviors can be used to suggest improvements to instruction that result in demonstrably better student learning [20]. These techniques have been successful in structured environments such as intelligent tutoring systems and have demonstrated applicability to many fields such as mathematics, vocabulary, and chemistry; however, with a few notable exceptions [1], they have seen less application in domains like programming.

In this work, we present a preliminary exploration of the application of knowledge-based learning curve analysis on programming data with the goal of extending the promise of educational data mining to programming. As this is, to our knowledge, the first attempt to apply learning curve analysis to programming data, our research questions are the following: can we successfully apply the methods of knowledge component modeling and learning curve analysis to code-writing programming data, and can we use the resulting models to evaluate student learning? In order to address these goals, we must first determine which individual concepts students might be struggling with. In this paper, we discuss multiple possible models of programming knowledge components, describe how we modified the traditional modeling process to be compatible with programming data, and analyze the resulting models, sharing our thoughts on the process along the way.

The contributions of this paper are:

- A modification to the traditional method of knowledge component modeling, to be used with code-writing data.
- Learning curves computed using real student data and categorizations of different syntax-based programming concepts based on these curves.
- Recommendations on future directions for knowledge modeling and learning curve analysis in the domain of programming.

2. Background
The work we present in this paper is rooted in the context of the Knowledge-Learning-Instruction (KLI) framework [13]. The KLI framework is concerned with providing a vocabulary for exploring how different types of knowledge constrain learning processes and in turn how different learning processes constrain which instructional choices will be optimal for robust student

learning. Central to the broader theory of KLI is the concept of a Knowledge Component (KC) which is defined as "an acquired unit of cognitive function or structure that can be inferred from performance on a set of related tasks." As learners are provided with instructional events for a particular KC it becomes more likely that they will demonstrate mastery of that KC when later assessed. This might seem straightforward, but it can be challenging to rigorously define what the KCs in a given domain are, and from there decide how best to structure the instructional environment to support those particular KCs.

On the other hand, most EDM research into programming data so far has focused on high level performance metrics and errors, instead of looking at how student code changes over time. However, there have been some exceptions. Several researchers have investigated the use of interaction networks as a way to interpret student work on a programming problem over time. This includes work on Parsons problems [7], Karel programs [16], and Java assignments [8]. These approaches tell us a great deal about how students develop their programs over time, but they do not separate out different knowledge concepts that can be investigated individually.

Some researchers have applied other EDM methods to programming data. Berges and Hubweiser used Item Response Theory (IRT) to compare the difficulty of different programming concepts, using concepts tagged on source code written by the students [3]. Kasurinen and Nikula applied Bayesian Knowledge Tracing (BKT) to programming data by predicting whether students would apply the correct structure to their code answers across sets of questions and found that about half of their students were predicted to have reached mastery on five central concepts by the end of their course [11].

In work closer to our own, Cherenkova *et al.* mapped problems to the concepts they were designed to test and used student success (on first attempt) to determine which concepts students were most challenged by [5]. They identified conditionals and loops as being particularly challenging for students. Yudelson *et al.* built student models based on code submissions over time, using the Rasch model and variations on the Additive Factors Model (AFM) [22]. Their process has many similarities to ours, but focuses on the question of how best to model students instead of investigating which specific concepts students are struggling with.

2.1 Learning Curve Analysis

Inspired by the power law of practice [14], learning curve analysis is an approach from broader EDM research that focuses on estimating learners' performance over time [4]. The learning theory behind the approach is that the probability that a learner would make an error in exercising a given skill should decrease over time as they get more opportunities to practice the skill. Traditionally, skills in this context are formalized as KCs, with a given task exercising one or more KCs. Students who possess a mastery of those KCs are more likely to perform the task correctly.

From a statistical perspective, learning curves are fit to student performance data using the Additive Factors Model (AFM) [4]. AFM is a specialized form of logistic regression that uses information about a student's prior practice opportunities with a set of KCs to predict the probability that they will perform correctly on a given opportunity. The mathematical formulation of AFM's regression equation takes the following form:

$$\ln\frac{p_{ij}}{1-p_{ij}} = \theta_i + \sum_k \beta_k Q_{kj} + \sum_k Q_{kj}(\gamma_k N_{ik})$$

This equation says that the log odds of a given student i performing step j, which exercises KC k, correctly can be predicted by a combination of an intercept for the student θ_i, an intercept for the KC β_k, and a KC slope γ_k., where N_{ik} represents a count of how many prior opportunities student i has had to practice KC k and Q_{kj} is a binary Q-matrix defining the mapping between KCs and steps.

The general assumptions of the AFM model are that every student possesses individual differences in initial ability, represented by the student intercept; every KC has a different initial difficulty, represented by the KC intercept; and that everyone tends to master a given KC at the same rate, represented by a single KC slope for all learners. In the most commonly used implementation of AFM [12] there is a fourth constraint imposed, not noted in the regression equation, where KC slopes cannot be negative (meaning that people do not unlearn or forget KCs).

3. Methodology

Before applying learning curve analysis to programming data we must first answer a few questions about how to adapt the usual methods to the unique properties of the programming domain. This is more difficult than it may at first sound, due to differences in the format of data traditionally used in learning curve analysis and the kind of data generated by programming tasks.

In the tutoring systems traditionally evaluated with learning curves, problems are broken down into individual steps, where a step is conventionally defined as the smallest unit of action that a student can perform correctly [21]. In programming data, there is no such definition of what a step should be. Programming data can be collected at as low a level as every keystroke made by students, but it is unclear how correctness could be measured for these low-level edits. Alternatively, data can be collected on student submissions and help requests; these submissions can be measured for correctness (using suites of test cases), but they also encompass many individual edits that have been made to the code.

In intelligent tutors, each problem's steps are tagged with one or more KCs at time of analysis. Since each step can be individually measured for correctness, each of the KCs can be analyzed for changes in correctness over time, to see whether students are learning. The analogy between tutor steps and submissions breaks down here, as failing test cases in a programming submission does not mean that every component of the code is wrong; it only means that a subset of the components are wrong. Therefore, we need to define a way to represent steps in programming problems that allows us to do useful data analysis.

To accomplish this, we address the following questions in the next sections:

- What are the KCs of programming?
- What are the steps and opportunities in a programming problem?
- How should correctness be measured for each of these steps?

3.1 Programming KCs

In order to model learners' acquisition of programming KCs, we first need to determine what the KCs of programming are, so that

we can construct a KC model that will accurately reflect student performance on programming tasks. In a broader sense this question has been a point of interest in the Computer Science Education Research community for many years, as many researchers have attempted to discover what the low-level concepts students are learning actually are.

First, one could view a programming problem as a set of constraints that the program needs to fulfill. These constraints can take the form of subproblems and/or test cases, which can all be individually assessed to see whether their requirements have been met. This model is a direct analogy to the step model typically used in tutoring systems. However, the subproblems and test cases are not themselves KCs, as we do not want students to learn how to solve specific test cases; we want them to learn how to write code that can be used to solve test cases like the ones assigned. Therefore, we still need to map these subproblems to sets of KCs.

Instead of using constraints, one could view programs through a broader algorithmic lens. From this perspective, students combine programming plans (common code substructures) in order to solve problems [19]. This approach has great potential as an accurate measure of student knowledge, but it assumes that students already understand the syntax of the programs they are writing, which is not always the case with novices. As we are investigating the work of new programmers in this paper, we leave algorithmic KC models for future work.

When considering the structure of knowledge in the programming domain, programming skill could be characterized as learning to choose the right program constructs (or combinations of constructs) in the appropriate circumstances for a specific goal. These can be represented as condition-and-response pairs, where a condition is the action that needs to be done and the response is the program token (or tokens) that can be used to execute that action. For example, if a student needs to store a value, they have to use the variable assignment operation. As a simplifying assumption, we could use the different textual tokens that appear in programs as indicators of construct use; additionally, if we can parse student code into Abstract Syntax Trees (ASTs), we can identify exactly when and where specific constructs are being used by walking the tree to find different node types. In this paper we utilize these AST node types to test whether this theory of programming knowledge as the ability to identify the correct conditions and provide the correct responses is an accurate model of programming KCs.

This approach still leaves some questions about implementation; for example, should each AST node type be treated as an individual KC, or should some tokens be collapsed together into broader categories? As a first step towards exploring this approach we decided to use a strategy that would include every token type in the built-in Python AST library. Once an initial candidate KC model has been created, it is common practice in learning curve analysis to explore various model refinements of merging more fine grained models, using both automated [4] and manual [20] methods; we plan to undertake this refinement in future work. These methods may help us determine how to change the original AST node types into KCs that are closer to the true knowledge that students are learning.

3.2 Programming Steps and Opportunities

A ‘step’ is hard to define in a generative context where students can do a lot of typing at once. For now, we choose to define a step at the level of a student’s deliberate action, whenever they submit a program or ask for a hint. However, we cannot use the submission process as a single opportunity for all of the problem’s KCs, since it implies that all KCs are required to be concurrently present in order for a student to solve a given submission correctly. While this is an accurate description of how the programming tutor provides feedback to students, it makes it difficult to tease apart which KCs students are struggling with most when they get a submission incorrect. Therefore, we say instead that each submission/hint request can be viewed as a single step that encompasses a sequence of opportunities in parallel, where each opportunity corresponds to an individual KC.

In traditional learning curve analysis, only the first attempt at each KC opportunity is used to measure student learning. This is done because traditional systems give students immediate feedback on each individual step. This means that after the first attempt, correctly solving a step does not necessarily mean that a student understands how to perform the step correctly; it could be that they are only doing what the feedback told them to, with no further comprehension. Programming problems seem different, as students are given feedback on the whole problem, not the individual tokens they’re writing. Still, it is possible that student might be applying the feedback to individual tokens. Therefore, we test two different kinds of KC step models: one which includes only data from the first attempt to each problem (where only the first submission of each session with a problem is counted in fitting learning curves), and one which includes all student submissions to the same problem. We call these two step-models First-Attempt and All-Attempts.

3.3 Step Correctness

The final question we must answer is how to determine if a particular application of a programming KC was done correctly. Since we are using AST node types as KCs, we can evaluate the correctness of each KC opportunity by determining whether that node has been used correctly in the student’s code. We define ‘correct use’ as follows: a) that the node occurs in the program, and b) that the node does not occur in the computed difference between the program and a correct version of the program, as generated by the ITAP algorithm [17]. In other words, an AST node type must be included in the correct portion of a student’s code to be measured as correct. For a given opportunity, we explore two alternative approaches to measuring correctness. First, we could say that every KC opportunity must be evaluated in every attempt. In that case:

- If the KC occurs in the edit between the student’s solution and the goal solution, it is INCORRECT
- If the KC is missing from the student’s solution and this is the student’s last (measured) attempt at that problem, it is INCORRECT
- Otherwise, it is CORRECT

Alternatively, we can say that all KC opportunities are evaluated in the first attempt, but in following submissions we only look at opportunities which changed after the previous attempt. Then:

- If the KC occurs in the edit between the student’s solution and the goal solution, it is INCORRECT
- If the KC occurs in the edit between previous and current state (or if this is the first attempt), it is CORRECT

- If the KC is missing from the student's solution and this is the student's last attempt at the problem, it is INCORRECT
- Otherwise, it is skipped for this step

Overall, we have a 2x2 variation structure with four KC models which we propose to build, as is shown in Table 1. We can think of the First-Attempts/Modified-Steps model as being closest to traditional KC models, and the All-Attempts/All-Steps model as providing the most data possible. We can compare these KC models to determine which provides the best learning curves, which we can then use to analyze student work and see what they understand and what they struggle with.

Table 1: The four KC models proposed in method modification.

First-Attempt/Modified-Steps	All-Attempts/Modified-Steps
First-Attempt/All-Steps	All-Attempts/All-Steps

4. Analysis

Now that we've hypothesized possible KC models, we need to test them with real student data to see whether they can produce viable learning curves. In the following sections, we describe how we generated the four models and prepared them for statistical analysis.

4.1 Dataset

The data we use comes from a study run in Spring 2016 on two introductory programming courses at Carnegie Mellon University. In this study, students were given access to an instance of Cloudcoder [15], an online IDE, which contained 40 Python programming practice problems covering a range of topics, including basic function structure, expression operations, conditionals, loops, lists, dictionaries, and recursion. Each practice problem had a Submit button which could be used to test the student's work against a collection of test cases, which would immediately showing the student the results. Students also sometimes had access to a Hint button which, when pressed, would generate a next-step hint for them based on the ITAP algorithm [18]. The study design randomized when students had access to the hint button, so that half of the students had access from weeks 1-3 of the study and the other half had access from weeks 4-6. All students could access the Hint button from weeks 7+.

We have not yet described how hint attempts would be included in the model, as they are not a typical component of programming log data. In traditional intelligent tutoring systems, hints are conventionally counted as an incorrect attempt at whatever KC the hint would have pointed the student toward. Therefore, we find the edit that was used to construct the hint (where the edit is composed of an old code snippet and a new code snippet), and for each KC opportunity determine whether the given node type occurred in the edit. Nodes which did occur are included as a HINT opportunity, while the other nodes are ignored. This process can be used across all four KC Model types, as it is a direct analogy from traditional ITS models.

All student use of the practice problems was optional, though students were told in class that completing practice problems could help them learn more. Out of 692 students in both classes, 89 agreed to having their data collected and chose to submit an answer for at least one programming problem. These 89 students made 2907 submissions and 380 hint requests over the course of the semester, resulting in a total of 3287 states over all 40 problems.

4.2 Model Generation Process

For each problem, we used Python's AST library to automatically identify all node types that occurred in the exemplar solution of the problem. For example, the problem *helloWorld* (which asked students to return the string 'Hello World!') contained five main AST node types: Module, Function Definition, Arguments, Return, and String. This is similar to the approach used in JavaParser to identify concepts occurring in Java programs [9]. 48 unique tokens were identified across all problems, with an average of about 11 tokens per problem.

We sorted the states by timestamp, then generated a solution space from all of the states using ITAP's path construction methodology [17]. For each state, we then used the solution space to identify the set of edits between the state and the closest goal state (for attempt states), or the specific edit that would be provided in a hint (for the hint request states). We identified the nodes that occurred in these edits and the original states, and the edits between original states and previous states by traversing the given ASTs.

For the Modified-Steps method, we generated a set of opportunities where opportunity names corresponded to node types and correctness depended on the method mentioned above (incorrect if node was not in the state or the goal-edit, correct if the node was in the state-edit, not included otherwise). For the All-Steps model, we used the same process, except that all KCs not marked as incorrect were marked as correct. The two models generated this way were both First-Attempt models; to generate the corresponding All-Attempts models, we went back through the steps and re-named them to include the step's iteration, as was described above. This resulted in all four models that we wished to test.

4.3 DataShop

At this point, we were ready to perform learning curve analysis on the KC models. To accomplish this we used DataShop, an online data analysis service [12]. In order to upload the models to DataShop, we generated files which included the following properties: Student ID, Timestamp, Student Response Type, Problem Name, Step Name, Outcome, and KC. (KC corresponded to the node type for each step, regardless of the used step name; the rest of the data was already available). DataShop automatically performed AFM on the datasets and generated learning curves for the KCs, making it possible for us to move directly to analysis of the results.

5. Results

Now we can examine the learning curves that resulted from this data in order to determine how well our models fit the traditional idea of a learning curve, and which KCs students struggle with the most. In all of the learning curves that follow, we cut off the graphs once the number of students included in each data point drops below 9, in order to make sure that at least 10% of our population is always represented (and to avoid strange outlier behavior).

First, we want to see what the whole-data-set learning curve (which averages the learning curves of all the KCs) looks like for each model. If the KC model is accurate, these curves should start with a high error intercept and then curve downwards, eventually

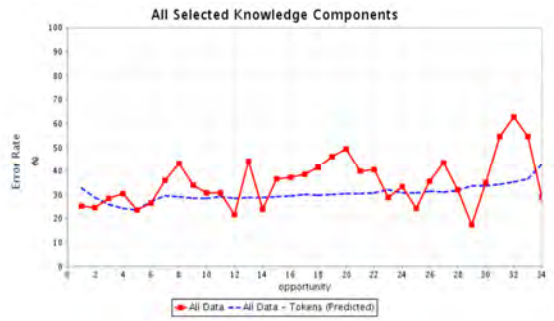
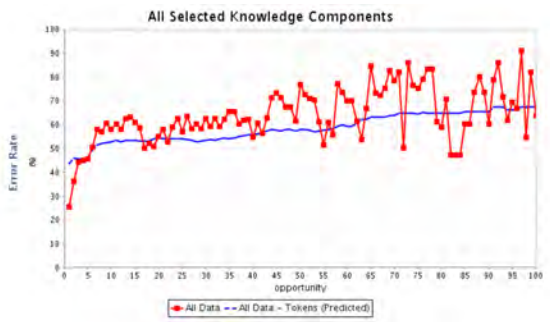
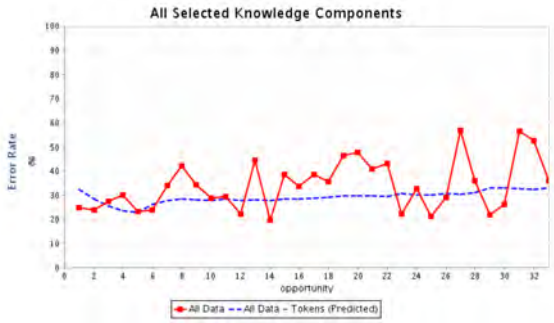
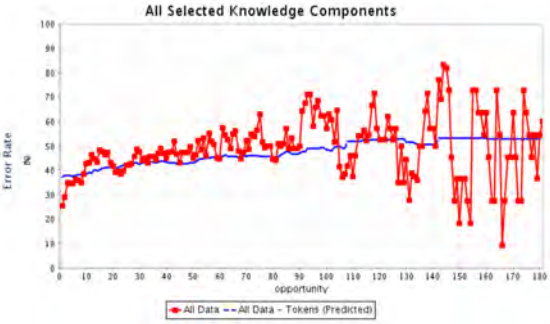
	First Attempt	All Attempts
Mod Steps		
All Steps		

Figure 1: Full-model learning curves for all four types of models.

plateauing near an error rate of zero (as students achieve mastery of all the individual KCs). As is shown in Figure 1, our KC models clearly do not match our expectations. The two All-Attempts curves actually increase over time, showing the opposite effect of what we expect based on theory. The two First-Attempt curves show more promise; they start with a declining curve over the first five states, then plateau at about 30% error rate afterwards. The data itself is jagged, but that does not necessarily invalidate our model as the curves for individual KCs can still have the expected downwards curve.

Comparing these results makes it clear that the First-Attempt model is a better fit than the All-Attempts model (which supports the traditional method of counting student attempts in tutoring systems). The Modified-Step and All-Step models, on the other hand, turned out to be almost identical in structure (due to the high overlap in their data), to the point that they could be used interchangeably. The Modified-Step model is closer to the traditional ITS model, so we will use it for the following analysis.

Next, we can examine the learning curves for each individual KC to determine where learning is happening. In DataShop, individual learning curves can be sorted into five categories: curves where there is not enough data 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 and show no downward trend (no-learning), curves that start and end with high error rates but do show a downwards trend (still-learning), and curves that start high but end low, demonstrating student mastery (good-learning). Optimally we'd like to see lots of good-learning curves, as they indicate that students are learning the concepts as expected, but the other curves are informative as well; already-learned concepts do not need to be covered as much, no-learning curves indicate problematic KCs, and still-learning curves demonstrate where extra practice is needed.

We examined the individual KC learning curves within the First-Attempt Modified-Step model, and found the following categorizations for our set of all AST node KCs:

- Little-data: !=, <=, >=, Alias, Dictionary, Divide, Expression, Import, In, List, Not, Not In, Or, Slice, Subtract, Tuple, Unary Operation, Unsigned Subtract, While
- 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

At first glance, this categorization only suggests that there are many KCs which are not being practiced enough (in the Little-data category) and many KCs which are not being learned (in the No-learning category). However, it's possible to glean much more information by investigating the individual learning curves and seeing what they show about the data. To demonstrate this, we share examples of the different kinds of learning curves (excepting little-data, which is usually non-informative), and we discuss what they might mean.

5.1 Good Learning Curves

First, we'll look at a successful learning curve for the Function Definition KC (as shown in Figure 2). This is an unusual KC as it only applied to the first six problems of the dataset that students

had access to; these problems were presented with no starter code, while all future problems were given with a function header (to help standardize student responses).

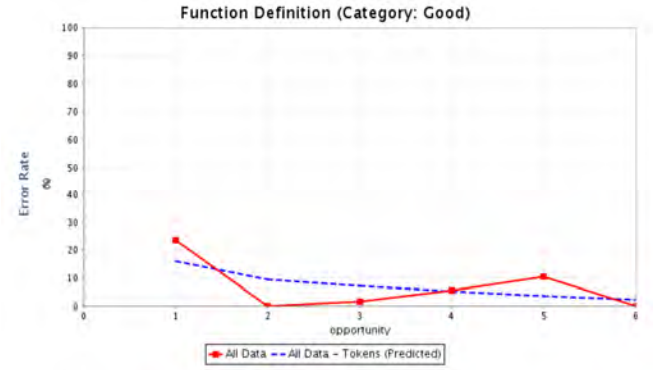


Figure 2: The good learning curve generated for the Function Definition KC. Starts at 23% error and approaches 0%.

In this learning curve, the error rate starts at an average of 23.4%; not as high as many other KCs, but high enough to indicate that around one out of five of students are struggling with the KC at their first attempt. In this model each new attempt is (usually) associated with a new problem; therefore, this graph shows that students struggled with the function definition in the first problem (helloWorld), but quickly mastered it in the following problems. We investigated the uptick at opportunity 5 and found that it was due to three of the twenty-five students asking for a hint before submitting anything to the associated problem (isPunctuation); as hints are counted as incorrect states by AFM, this resulted in an increase in the error rate.

The Arguments learning curve is very similar to Function Definition, which is sensible as the two occur together. However, the Compare learning curve (shown in Figure 3) demonstrates a different learning effect. The data in this model mostly progresses slowly downwards, but it has a blip at opportunity 5, where it hits 0 only to jump back up again. This seems to be due to a problem which used comparison very simply (to check if a value was less than 0, an edge case) that is surrounded by problems which use multiple comparisons that must be combined. This might be a sign that the skill required to use a single comparison operation is different from the skill of combining multiple comparisons, and that the two should be separated into different KCs (and taught as separate concepts as well).

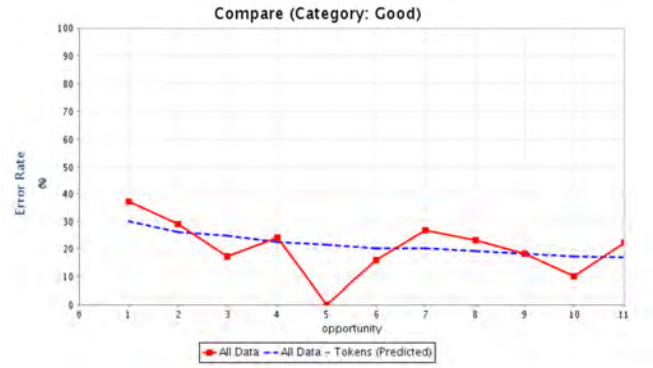


Figure 3: The learning curve generated for the Compare KC. The model trends downwards, from 30% to below 20%, and the data shows a similar trend.

5.2 Still-Learning Curves

Next we'll look at a learning curve which seems to be trending downwards (as we'd expect), but does not reach a low enough error rate to say that the student has truly learned the concept. The Call KC occurs every time the student uses any function call in their code; therefore, it is surprising to see that this data is mostly consistent (albeit with a jagged appearance) and steadily heading downwards. This suggests that students are not learning new concepts for every new function they have to call; instead, they are learning a single concept, how to use *any* function call successfully. With more opportunities, we would expect this model to reach mastery. Therefore, we may wish to include more problems that let students practice using function calls.

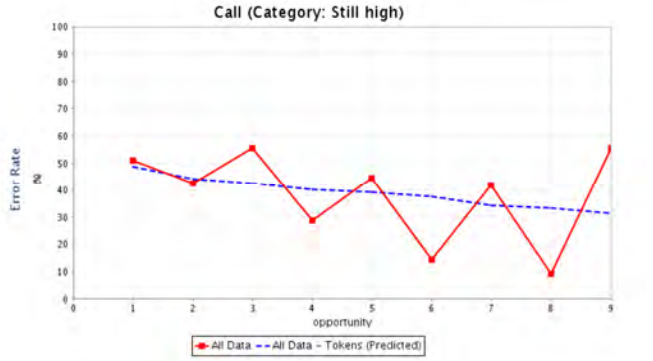


Figure 4: A learning curve that has not yet demonstrated mastery, showing student progress in the Call KC. The model starts at around 50% error rate and reaches 30% error rate before running out of opportunities.

5.3 Already-Learned KCs

One of the learning curves generated by our method was quite surprising to us; we did not expect to find that If statements would be classified as already-learned (as is shown in Figure 5)! This classification could be due to several reasons; perhaps the students are using if statements correctly but making errors in the tests and bodies of the statements, or perhaps the concept of a conditional is intuitive enough that students truly do solve the problems with no trouble. To investigate this, we looked more closely at the data associated with this learning curve, and we quickly determined that the real cause might be the method used to generate KC models at the beginning.

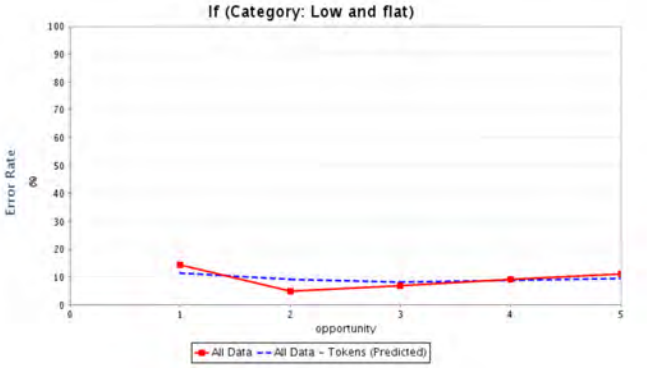


Figure 5: The learning curve for the If Statement KC, which seemingly demonstrates that students have mastered using if statements from the very beginning. The curve remains below 20% error for all opportunities.

For each problem, we had generated a set of KCs to be measured based on the teacher solution to the problem. In the teacher view of the problems, many of the early problems could be solved by returning simple boolean expressions. However, students who solved these problems often opted to use conditionals instead, often writing code of the form *if (boolean expr): return True else: return False*. This means that students often got a great deal of practice at writing conditionals that was not actually measured. If these problems included If statements in their KC models, we would expect a more accurate learning curve for those students; however, this would also unfairly penalize the students who did not use If statements but succeeded in solving the problem. This could possibly be remedied by basing the KC model for each problem and each student off of the student's eventual correct state, instead of the teacher's goal state; however, this would provide less overlap between student models, and would provide less connection to the teacher's intended problem design.

5.4 No-Learning Curves

Now that we have covered what the successful KC's learning curves look like, we can start examining what might be happening with the curves that show no learning at all. Upon examination, these fall into two categories: data which has a flat error rate, and data with more jagged error rates.

The first category (flat error rate) is demonstrated by Boolean Operation, For, Index, and Subscript. We'll use the For KC as an example here (shown in Figure 6). This curve is exactly what we'd expect: it demonstrates a consistent error rate, which, though low, is not good enough to reach mastery. Investigation into the individual opportunities did not show a consistent reason for this; some error states were due to missing for loops, while others were caused by for loops being used in the wrong place.

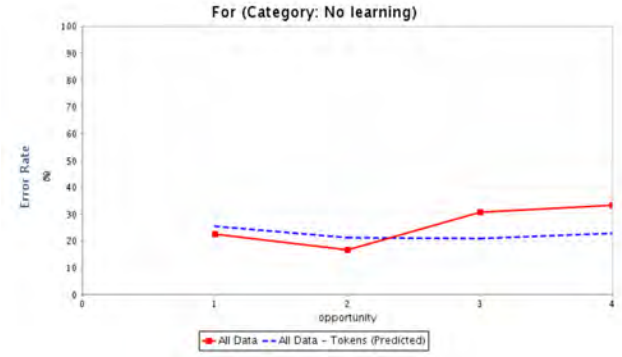


Figure 6: The learning curve for the For Loop KC, which shows a flat error rate. The error rate stays consistently between 20 and 30%.

The data with jagged error rates is more interesting to examine. Attribute, Binary Operation, and Return give us this effect; we'll examine the Binary Operation KC (which encompasses all non-comparison or boolean operations with two operands), shown in Figure 7. By investigating the incorrect states that are represented by each of the opportunities, we can determine why students do so well at some states and so poorly at others.

First, we looked into the low error rate opportunities (1, 2, 3, 5, 9, and 12). The first four were all problems where the only goal was to perform mathematical operations which were provided by the prompts, and it seems that most students had already mastered these mathematical skills (which is not surprising, as these are similar to basic calculator operations). The other opportunities, with higher error rates, have a range of causes: some used non-mathematical operations (like concatenating strings), and some combined multiple operations in non-intuitive ways. Furthermore, binary operations such as x+1 were often treated as basic values in the more complex programs, and thus could often be marked incorrect when the real blame fell on higher-order constructs. This also occurred with the Name, Number, and String KCs; all had wildly inconsistent learning curves due to their prolific use throughout programs.

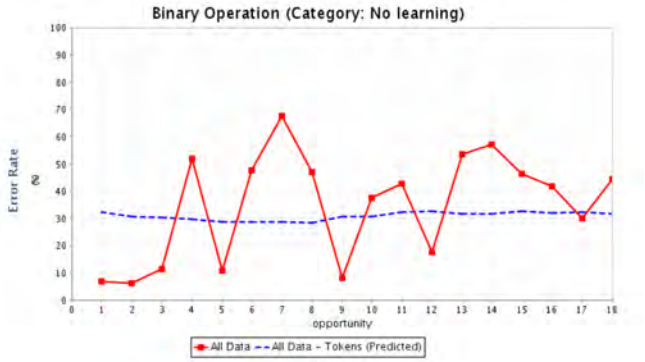


Figure 7: A learning curve generated for the Binary Operation KC. The jagged pattern indicates potential problems with the KC.

It may be possible to improve the fit of binary operations by splitting them up based on the operator type; however, the learning curves generated for the operators themselves either have too little data to identify learning or exhibit no learning at all. We might meet more success if we group up the operators into sub-categories and separate out the different purposes that some of the overloaded operators (like addition) have.

6. Discussion

In this paper, we've attempted to automatically generate KC models for programming out of the AST components used in program solutions. We've investigated whether these models should be generated using all attempts vs the first attempt only, and whether only modified nodes should be included in each opportunity vs all KC nodes. Our results show that the first attempt model tends to produce cleaner learning curves, and the two step models perform equally well. However, the general learning curves produced do not show the expected reduction in error rate. To determine why, we looked into the individual KC learning curves.

In investigating these individual learning curves, we found wide-ranging results. First, we found that several KCs did not contain enough data to demonstrate learning; in other words, the students weren't getting enough practice to master those concepts. This can be remedied by producing more problems that cover the neglected KCs. Many KCs did not exhibit any learning (possibly due to bad KC modeling), but some did show learning, either complete or partial. Investigating the erroneous states that led to these learning curves helped us identify several unexpected occurrences, such as the fact that students had already mastered If statements by the time they reached the problems where they were first supposed to use them, and the fact that function calls seem to be learned independently of different function types. Findings such as these, if they are validated in future work, could be used to inform the teaching of programming (for example, by including the instruction of conditionals much earlier in the

curriculum, as students will apparently use them regardless of instructor intent).

In future work, we plan to modify the KCs used in the models we've generated to see if we can improve the modeling of the student data. Several possible modifications were mentioned in the Results section; for example, we might group similar AST nodes (such as the comparison operators <, >, <=, and >=) together into joint KCs, as they can all be used for semantically equivalent purposes. We might also try to identify when a single KC is actually representing several different concepts; for example, we can split the Add operator into its different types (string concatenation and numeric addition), and we can try to identify simple edge-case comparisons and separate them from more complex comparisons. Modifying these KC models may lead to better-fitting learning curves that better model how students are learning.

We've also considered additional approaches that could be used in defining programming step opportunities and correctness in the processing stages of model construction. First, we could generate KC models for each problem based on the student's individual goal state, instead of the teacher goal state; this would allow us to capture learning on all the AST nodes that a student used, instead of just measuring learning on the originally intended concepts. This approach might remedy the problem shown in the If statement KC. Alternatively, instead of using path construction to identify the correctness of KCs, we could use the test cases assigned to the program, where each test case could be mapped to the set of nodes which that test case is supposed to cover. This would require more advanced program analysis, but could yield more accurate results without relying on the creation of goal states for correctness measurement. We also considered using the provided teacher solution for comparing student solutions to a goal state instead of computing goal states via algorithm; however, attempting this quickly revealed that it was a useless approach, as the multitude of possible solutions led to an artificially high error rate in all KCs.

In this paper we demonstrated how DataShop could be used to examine learning curves and investigate erroneous states; however, there are other DataShop features which we did not utilize. For example, the Performance Profiler feature can be used to identify the error rates for different KCs across different problems/steps/etc. Using this system, we could look more deeply into the data to determine whether some KCs are being used in different contexts at different times (for example, if the Add node has different error rates when adding numbers when compared to adding strings or lists). This could help us start building up more semantic KC models which may have a greater chance at successful modeling.

There are several limitations to our work. First, we are using learning curve analysis and AFM in ways that they were not originally designed for; it is possible that the results we have gotten from them may not be indicative of actual student learning. Additionally, our decision to define KC correctness based on the edits between the state and a chosen goal state is a very rough approximation of true KC correctness. It's possible that the optimal goal for the student's current state will not have been chosen, resulting in more edits than are necessary; it's also possible that a node could be correctly used, but would still need to be changed in order to get to a correct code state. Future work in hand-coding the states might be used to see how accurate this automated approach is at estimating KC correctness.

Furthermore, our method of representing steps as simultaneously-executed actions is very different from the traditional implementation of steps in intelligent tutoring systems and other educational technology. Compounding this is the fact that there may be many other invisible steps that we are not measuring at all, such as students' design decisions, which occur before they even start to write code. We welcome suggestions on how this step model could be modified to better represent the reality of student work.

In traditional intelligent tutoring systems, it is assumed that problem sequencing is handled with a mastery paradigm [6]; once a student has mastered a concept the system can move on to a new topic. However, in a conventional programming task, this is impossible, since old KCs must be used to build up new ones as more advanced concepts are learned. Therefore, it's possible that the behavior of learning curves will be different in this context than in other, less construction-based contexts.

Finally, we were unable to construct a validation strategy for the models at the time of writing, as we did not yet have student performance data outside of the system to compare the models to. Therefore, we must rely on the success of AFM in other domains as evidence that the models can truly emulate student learning. If student data was available, we could check for similarity between the produced model's student intercepts (which are supposed to simulate the incoming ability of individual students) and the pretest scores of students in the class; this at least could be used to determine whether the model as a whole mirrors reality accurately. Additionally, we could design test items to target specific KCs, then have the students complete these items after practicing, in order to see whether students perform better on items that are shown as mastered in the model. We hope to use these approaches and others in future work.

7. Conclusion

We hope that, overall, this work can serve as evidence of the fact that programming data can be evaluated using approaches that more closely examine the code that students produce and the learning that students do over time. Approaches such as KC modeling and learning curve analysis can help us understand the precise concepts that students are struggling with, which may inform the future design of programming curricula in order to better enable learning. There are many modifications that could be made to the models presented here in order to more accurately represent programming knowledge, and we hope that others will use and adjust some of the method presented in this paper to test these modifications in future work.

8. Acknowledgements

Thanks to Jason Imbrogno for his help with an early version of this project. This work was supported in part by Graduate Training Grant awarded to Carnegie Mellon University by the Department of Education (# R305B090023).

9. References

[1] Anderson, J.R. and Reiser, B.J. 1985. The LISP Tutor. *BYTE*. 10, 4 (1985), 159–175.

[2] Baker, R.S.J.D. and Yacef, K. 2009. The State of Educational Data Mining in 2009 : A Review and Future Visions. *Journal of Educational Data Mining.* 1, 1 (2009), 3–16.

[3] Berges, M. and Hubwieser, P. 2015. Evaluation of Source

Code with Item Response Theory. *ITiCSE '15* (2015), 51–56.

[4] Cen, H. et al. 2006. Learning Factors Analysis – A General Method for Cognitive Model Evaluation and Improvement. *ITS '06* (2006), 164–175.

[5] Cherenkova, Y. et al. 2014. Identifying Challenging CS1 Concepts in a Large Problem Dataset. *SIGCSE '14* (2014), 695–700.

[6] Corbett, A.T. and Anderson, J.R. 1995. Knowledge Tracing: Modeling the Acquisition of Procedural Knowledge. *User Modeling and User-Adapted Interaction*. 4, 4 (1995), 253–278.

[7] Helminen, J. et al. 2012. How Do Students Solve Parsons Programming Problems? — An Analysis of Interaction Traces. *ICER '12* (2012), 119–126.

[8] Hosseini, R. et al. 2014. Exploring Problem Solving Paths in a Java Programming Course. *PPIG '14* (2014).

[9] Hosseini, R. and Brusilovsky, P. 2013. JavaParser: A Fine-Grain Concept Indexing Tool for Java Problems. *AIEDCS '13* (2013), 60–63.

[10] Ihantola, P. et al. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. *ITiCSE WG '15* (2015), 41–63.

[11] Kasurinen, J. and Nikula, U. 2009. Estimating Programming Knowledge with Bayesian Knowledge Tracing. *ITiCSE '09* (Aug. 2009), 313–317.

[12] Koedinger, K.R. et al. 2010. A data repository for the EDM community: The PSLC DataShop. *Handbook of educational data mining*. 43.

[13] Koedinger, K.R. et al. 2012. The Knowledge-Learning-Instruction Framework: Bridging the Science-Practice Chasm to Enhance Robust Student Learning. *Cognitive Science*. 36, 5 (Jul. 2012), 757–798.

[14] Newell, A. and Rosenbloom, P.S. 1981. Mechanisms of skill acquisition and the law of practice. *Cognitive skills and their acquisition*. 1–56.

[15] Papancea, A. et al. 2013. An Open Platform for Managing Short Programming Exercises. *ICER '13* (2013), 47–51.

[16] Piech, C. et al. 2012. Modeling How Students Learn to Program. *SIGCSE '12* (2012), 153–158.

[17] Rivers, K. and Koedinger, K.R. 2014. Automating Hint Generation with Solution Space Path Construction. *ITS '14* (2014), 329–339.

[18] Rivers, K. and Koedinger, K.R. 2015. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education [pre-release]*. (2015).

[19] Soloway, E.M. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*. 29, 9 (Sep. 1986), 850–858.

[20] Stamper, J.C. and Koedinger, K.R. 2011. Human-Machine Student Model Discovery and Improvement Using DataShop. *AIED '11* (2011), 353–360.

[21] VanLehn, K. et al. 2007. What's in a Step? Toward General, Abstract Representations of Tutoring System Log Data. *UMAP '07* (2007), 455–459.

[22] Yudelson, M. V. et al. 2014. Investigating Automated Student Modeling in a Java MOOC. *EDM '14* (2014), 261–264.