# Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming

**Article** *in* Journal of the Learning Sciences · November 2014
DOI: 10.1080/10508406.2014.954750

**6 authors**, including:

Paulo Blikstein
Stanford University
**144** PUBLICATIONS **1,629** CITATIONS
SEE PROFILE

Marcelo Worsley
Northwestern University
**32** PUBLICATIONS **317** CITATIONS
SEE PROFILE

Chris Piech
Stanford University
**15** PUBLICATIONS **880** CITATIONS
SEE PROFILE

Daphne Koller
Stanford University
**408** PUBLICATIONS **41,284** CITATIONS
SEE PROFILE

Journal of the Learning Sciences
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/hlns20

# Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming

Paulo Blikstein[a], Marcelo Worsley[b], Chris Piech[c], Mehran Sahami[c], Steven Cooper[c] & Daphne Koller[c]
[a] School of Education and (by courtesy) Computer Science Department, Stanford University
[b] School of Education, Stanford University
[c] Computer Science Department, Stanford University

Accepted author version posted online: 04 Sep 2014.Published online: 24 Oct 2014.

To cite this article: Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper & Daphne Koller (2014) Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming, Journal of the Learning Sciences, 23:4, 561-599, DOI: 10.1080/10508406.2014.954750

To link to this article: http://dx.doi.org/10.1080/10508406.2014.954750

THE JOURNAL OF THE LEARNING SCIENCES, 23: 561–599, 2014
Copyright © Taylor & Francis Group, LLC
ISSN: 1050-8406 print / 1532-7809 online
DOI: 10.1080/10508406.2014.954750

# Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming

Paulo Blikstein
*School of Education and (by courtesy) Computer Science Department*
*Stanford University*

Marcelo Worsley
*School of Education*
*Stanford University*

Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller
*Computer Science Department*
*Stanford University*

New high-frequency, automated data collection and analysis algorithms could offer new insights into complex learning processes, especially for tasks in which students have opportunities to generate unique open-ended artifacts such as computer programs. These approaches should be particularly useful because the need for scalable project-based and student-centered learning is growing considerably. In this article, we present studies focused on how students learn computer programming, based on data drawn from 154,000 code snapshots of computer programs under development by approximately 370 students enrolled in an introductory undergraduate programming course. We use methods from machine learning to discover patterns in the data and try to predict final exam grades. We begin with a set of exploratory experiments that use fully automated techniques to investigate how much students change their programming behavior throughout all assignments in the course. The

Correspondence should be addressed to Paulo Blikstein, Stanford University, School of Education, 520 Galvez Mall—CERAS 232, Stanford, CA 94305. E-mail: paulob@stanford.edu

Color versions of one or more of the figures in the article can be found online at www.tandfonline.com/hlns.

results show that students' change in programming patterns is only weakly predictive of course performance. We subsequently hone in on 1 single assignment, trying to map students' learning process and trajectories and automatically identify productive and unproductive (sink) states within these trajectories. Results show that our process-based metric has better predictive power for final exams than the midterm grades. We conclude with recommendations about the use of such methods for assessment, real-time feedback, and course improvement.

Scholars have been advocating for the benefits of student-centered, inquiry-based, and project-based learning environments for decades (Dewey, 1902; Freire, 1970; Montessori, 1964, 1965; Papert, 1980). Although this vision has captured the hearts and minds of educators since the turn of the 20th century, its need and feasibility has come under attack several times during the past decade (e.g., Kirschner, Sweller, & Clark, 2006; Klahr & Nigam, 2004). In recent years, however, because of the transformed societal and economic environment and the urgent need for higher level, complex problem-solving skills (Levy & Murnane, 2004), the need for those new approaches has made a strong comeback both at the K–12 level (e.g., Barron & Darling-Hammond, 2010; Blikstein, 2013) and in engineering education (Dutson, Todd, Magleby, & Sorensen, 1997; Dym, 1999; Dym, Agogino, Eris, Frey, & Leifer, 2005). In addition, the rapid growth of massive open online courses has increased the need for scalable pedagogies that go beyond the lecture and for automated assessments that go beyond the multiple-choice tests.

We argue that developing new automated data collection and analysis techniques, rather than automating and scaling up the outdated, behaviorist-inspired teaching and assessment approaches that have dominated educational institutions, could offer new, scalable opportunities to advance student-centered, project-based learning. More fine-grained data collection and analysis techniques might help advance constructivist approaches by revealing students' detailed trajectories throughout a learning activity, helping designers identify better scaffolding strategies, alleviating assessment bottlenecks in large-scale implementations of project-based learning, offering rich real-time feedback that allows practitioners to tailor their instruction, and providing unprecedented insight into students' cognition (Berland, 2008; Blikstein, 2009, 2011a, 2011b, 2013; Roll, Aleven, & Koedinger, 2010). These techniques could offer novel insights into learning, especially when students have ample space to generate unique solutions to a problem, which is common in activities such as building a robot or programming a computer. What insights could we gain by increasing the rate, detail, and automaticity of observation while using machine-learning algorithms to find patterns in these complex data? Computer programming is an especially appropriate task for testing these methodologies for three reasons. First, researchers have noted that computer programming can itself be regarded as a record of students' cognitive processes (Papert, 1980, 1987; Pea, Kurland, & Hawkins, 1987). Second, collecting snapshots of students' work is technologically unproblematic. Third, there has been much interest in promoting computational thinking in K–16 education (National Research Council, 2012; Wing, 2006), researching various strategies and curricula for the engagement of children in learning to program (diSessa, 2000; Kafai, 2006; Maloney, Peppler, Kafai, Resnick, & Rusk, 2008), as well as developing new programming languages for education. Given that computer programming is becoming more popular in schools and after-school programs, we hope that our work will have wide applicability in K–16 education for a range of age groups and types of programming languages.

In this article we use an automated system to capture snapshots of students' code during programming assignments throughout an introductory undergraduate course in programming methodology and then use machine-learning techniques to track students' progression. We show that patterns and commonalities can be identified even within the highly personal trajectories of hundreds of students and that those patterns can both illuminate students' programming behaviors and predict their future performance on exams.

Our data consist of more than 154,000 code snapshots captured from the assignments of 370 students. The first set of experiments looks for patterns across several assignments and tries to correlate those patterns with students' assignment and exam grades. The second study examines one single assignment in depth and attempts to build machine learning–induced progression maps, showing that the topologies of such maps are correlated with course performance.

## PREVIOUS WORK

There have been several efforts to assess student learning in computer programming (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011; Ioannidou, Bennett, Repenning, Koh, & Basawapatna, 2011). In this section, we situate our work within this landscape by outlining paradigms that range from outcome-based to process-based assessments. Seminal work on microgenetic methods (Schoenfeld, Smith, & Arcavi, 1991; Siegler & Crowley, 1991) and microethnographies (Nemirovsky, 2011) has demonstrated the value of greatly increasing the frequency and level of detail in the analysis of learning trajectories. Educational data mining and learning analytics methods (Aleven, Roll, Bruce, & Kenneth, 2010; Baker, Corbett, Roll, & Koedinger, 2008; Baker & Yacef, 2009; Roll, Aleven, McLaren, & Koedinger, 2011) now promise to enable the utilization of even higher frequency data and enable the discovery of deep patterns in the data that would otherwise remain unobserved, allowing for richer examination of *processes* instead of *outcomes.* Automated outcome-based assessments in computer science require very little human intervention, but they ignore process. Process-based assessments, however, are typically mediated through direct human observation and are difficult to scale. Our work is intended to leverage the affordances of both ends of this spectrum, and thus we present a collection of prior work about these two types of assessments.

## Assessment in Computer Science Education

Traditional approaches to the automated assessment of programming tasks have focused on outcomes (Cooper, Cassel, Cunningham, & Moskal, 2005; Fuller et al., 2007). A student's program is automatically run against a battery of test cases to analyze its correctness and efficiency (College Board AP, n.d). Despite the efficiency of these methods for grading students, researchers have tried to emphasize that programming is not just the ability to generate code; it may also be regarded as a way of thinking, decomposing, and solving problems ("Computing Curricula 2001"; Marion, Impagliazzo, St. Clair, Soroka, & Whitfield, 2007). When trying to analyze these other goals, researchers have found unclear or counterintuitive interactions between prior programming knowledge and performance in computer science courses, as well as students' knowledge and their ability to apply it to real-world problems, so there are several open questions about the actual constructs being measured when students engage in traditional assessments in computer science courses.

For example, Simon et al. (2008; Simon, Chen, Lewandowski, McCartney, & Sanders, 2006) and VanDeGrift et al. (2010) examined the notion of "being" and thinking like a computer scientist by investigating students' ability to apply computer programming to real-world problems (common sense computing). They challenged learners to utilize a technique from computer science, such as sorting or search, to complete an everyday task. Simon et al. (2006) found that even before beginning formal computer science instruction, students have some knowledge of core constructs such as looping and sorting, but they do not understand or utilize these concepts as experts would. As a result, they observed a *decrease* in student performance on real-world computing tasks after the completion of an introductory computer science curriculum. In the realm of debugging, a similar phenomenon was observed by Simon et al. (2008), who reported that even though students exhibited preexisting expert-level skills for troubleshooting problems (such as examining the structure of problems and doing incremental testing), they were not able either to efficiently identify problems or to evaluate when a strategy should be abandoned. Similar results about the knowledge of logic were reported by VanDeGrift et al.

Counterintuitive findings of this kind prompted the research community to consider that the problem was more intricate, and this has prompted scholars into a more fine-grained and process-oriented analysis of student work. Soloway and Ehrlich (1984) pioneered much of this field through their work on the programming strategies of novices and experts. In their analysis, they typified experts as possessing two primary characteristics: (a) the ability to plan a program and (b) sufficient syntactic knowledge to implement their plan. Soloway and Spohrer (1988) later extended this approach through bug logging, which logged program errors. Even though their analysis did not exhaustively capture students' code snapshots, the system allowed them to monitor the type and quantity of errors that students encountered. Results showed that the majority of bugs were indicative of planning errors as opposed to lack of knowledge about the programming language. This study was an early indication of the impact of planning on students' programming processes. Much later, Jadud (2006) introduced the idea of looking at successive compilations in order to track the progression of a program. He focused mostly on the types of syntactic errors that students generate and found that most were relatively simple, but when they occurred in large numbers, they significantly reduced the time available for students to study more essential components of programming. By pointing to these elements, Jadud was one of the first to systematically categorize errors and challenges that students run into in introductory programming courses. However, he could not study more elaborate behaviors because he only had access to data from a very limited number of students.

Blikstein (2009, 2011b) utilized thousands of complete code logs drawn from nine subjects who were engaged in a 4-week programming assignment. He proposed a set of novel techniques for capturing code snapshots over extended assignments and the use of automated techniques to analyze them. His focus was to identify salient aspects of expert and novice programming styles, uncovering unique logical and stylistic elements. Results showed that more experienced students were more likely to adopt an incremental coding strategy (trying to debug and advance their code without external help through myriad trial-and-error attempts), whereas novices would update their code in larger batches, copying and adapting code from sample programs and other external sources. He also demonstrated that different stages in the programming process would exhibit distinct compilation frequencies and types of error messages.

Curiously, a common approach that emerged in the field was to categorize students into several types based on their behavior or debugging pattern. Perkins, Hancock, Hobbs, Martin, and Simmons (1986), for example, investigated students in a laboratory environment programming in BASIC. They typified their subjects according to the simple dichotomy of movers and stoppers. Booth (1992), Cope (2000), and Bruce et al. (2004), instead of using students' code, used phenomenography and facilitated programmers to vocalize their internal experience while programming. Based on students' patterns of participation and learning, they identified five programming pathways typically followed by students: followers, coders, understanders, problem solvers, and participators.

But the most well-known attempt to systematize categories of programmers is framework defined by Turkle and Papert (1992), which proposed the existence of *tinkerers* and *planners.* Different from most of the previous research, their goal was not simply to create categories but to put forth the idea of epistemological pluralism and show that both groups of students could achieve at a high level while taking different paths and approaches. Papert and Turkle attempted to show that even though we would conventionally consider the planning behavior as superior, planning and tinkering should not be seen as desired behaviors, but just diverse ways of approaching the same problem without necessarily achieving superior or inferior results. More recently, other researchers tried to find these categories and validate these initial ideas in much larger data sets using data mining techniques (Berland & Martin, 2011). They clustered large numbers of student logs and found that most successful students followed either the tinkerer or the planner approach. Tinkerers make a series of incremental changes to their programs in order to create a finished solution. Planners identify a course of action and implement a plan to systematically produce their final program submission.

The tinkering and planning categories made their way outside of the education research. Researchers in human–computer interaction have also investigated them. Beckwith et al. (2006) and Burnett et al. (2011) conducted studies on how males and females differ in their strategies for fixing bugs in programmable spreadsheet software. The main findings of these two studies were that women tinkered much less than men, which in turn caused them to introduce more bugs into the system. Women also did not use the unexplained features of the system as frequently as men did. The authors showed that such categories of behaviors are important and can have an impact on performance—namely, counterintuitively, tinkering was an important determinant of performance because it allowed users to find new features in the system. After a research-based training program (which addressed gender self-efficacy), the authors found that the difference was greatly reduced in both studies. Tinkering and planning behaviors in human–computer interaction were also studied by Fern, Komireddy, Grigoreanu, and Burnett (2010), who used machine-learning techniques in place of traditional statistical methods to treat, filter, and cluster multidimensional data sets. Their algorithms categorized users' actions into typical microstreams of five or six consecutive actions and examined measures such as the frequency and ratio of successful to unsuccessful actions.

The fact that so many researchers ended up using categories to describe how students engage in programming reveals a tacit recognition of the importance of epistemological pluralism. Perhaps because programming was a relatively new area in educational research, scholars were more open to observing these different pathways. One current concern would be that as computer science education becomes more widespread and mainstream, this recognition of these multiple ways to learn would give place to monolithic curricula, as we observe in many other disciplines.

We found, however, gaps in the literature—little research exists on the development and study of scalable process-oriented approaches. We identified three main problems in this area: (a) Most studies, with the exception of the very recent ones by Berland and Martin (2011) and Fern et al. (2010), were too small, relying on manual coding and observations; (b) larger studies did not capture data at the level of detail required to build complex models of students' programming process; and (c) the tasks proposed to students were quite constricted, often in the form of laboratory studies rather than real-world programming assignments. In our present work, we address this gap in the research through the simultaneous study of *open-ended tasks* and *large numbers of students* and the application of powerful new *process-based, machine-learning techniques.*

Expanding the methodology of Blikstein (2009, 2011a, 2011b), who designed schemes to capture, filter, and analyze code snapshots, we instrumented the programming environment used by students (the Eclipse platform) to capture and record complete snapshots of students' code whenever they saved or compiled their programs (Piech, Sahami, Koller, Cooper, & Blikstein, 2012).

Using machine-learning techniques, we then demonstrate that students' programming paths contain robust commonalities and patterns, and these patterns may be used predictively to infer students' subsequent programming steps and performance in the course. We conduct two types of explorations: The first is a series of simpler experiments about the relationship between patterns of code update and course performance, with a special focus on how these patterns change over several assignments. In these first experiments, we only look at how many lines of code were added or modified, without looking at the content of the code. In Study 2, we delve deep into the *content* of the code snapshots. Using a variety of machine-learning techniques, we transform the myriad code snapshots into maps of states that show the progress of the students' work and the correlation of this work with student performance.

Because the algorithms and data collection schemes used in this article may be unfamiliar to some members of the learning sciences community, we provide more detailed explanations of all of them in the supplemental material.

## PRELIMINARY EXPLORATIONS: CAN LEARNING ANALYTICS TECHNIQUES DETECT PATTERNS IN STUDENTS' TRAJECTORIES OVER SEVERAL ASSIGNMENTS?

Our approach in this section is to develop an increasingly complex exploratory analysis of the data, starting from the simplest possible type (regression of aggregate data). The results (or lack thereof) guide the next step up in complexity:

further disaggregation of the data, and other methods (i.e., clustering). There are two goals for this approach: The first is the search of possible low-hanging fruits in our analysis, given the high computational cost of some machine-learning algorithms. The second goal is to illustrate how a large learning data set can be explored using a series of simple investigations of increasing complexity. This second goal could be particularly useful to the learning sciences community, as we start from well-known statistical techniques and move toward more elaborate analysis in small steps.

In this first series of experiments, we were interested in general purpose, fully automated techniques that would reveal patterns and meaningful trajectories in our raw data without human labeling (which is labor intensive) or any analysis of the actual content of students' code (which requires considerable computational power). Rather than searching for predictive models of student performance, we simply asked the question, Can relatively simple quantitative measures capture differences in students' behaviors when programming, and are those related to course performance? Then in Study 2 we used machine-learning techniques of much higher complexity that looked into the content of the code.

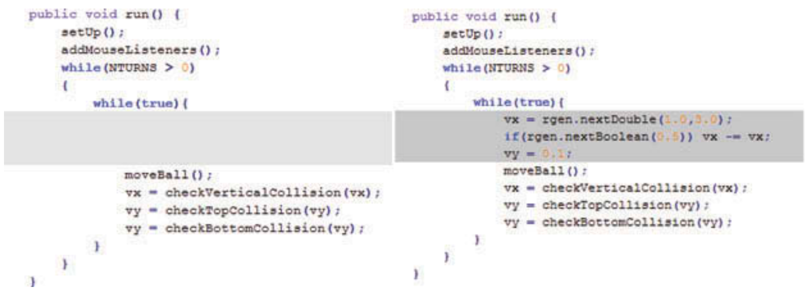
Therefore, when considering these explorations, the reader should bear in mind that our main goal is to explore initial metrics that may enable future research on programming patterns, or at least reveal research paths that might *not* be worth pursuing. Given that this is the beginning of trying to use this type of process data in education, we consider that it is important to document even very early attempts to create these metrics so that they can be improved upon.

The main thrust of our explorations into metrics of students' programming patterns is the study of the two hypothesized behaviors defined by Papert and Turkle (1992): tinkering and planning.

## General Methodological Notes and Dimensions of Analysis

We attempt to measure students' behaviors in three experiments, which always begin with the comparison of two consecutive code snapshots and the determination of how much change has taken place from one snapshot to the next as measured by the number of *characters* or *lines of code* that students have added, removed, or modified. Thus, for each code snapshot generated by a student, we calculate a set of six measures: *number of lines added, lines deleted, lines modified, characters added, characters removed,* and *characters modified.* We call this set of measures the *code update differential.* These measures exclude comments and are based on computing the line-by-line difference between snapshots.[1] Our

[1]Modification of a line was defined to have taken place anytime a new line was 30% different from the same line in the previous snapshot. The designation *characters modified* refers to the absolute value of the difference between the original and the modified lines. The 30% value was selected to strike a



```
public void run() {
    setUp();
    addMouseListeners();
    while(NTURNS > 0)
    {
        while(true){



            moveBall();
            vx = checkVerticalCollision(vx);
            vy = checkTopCollision(vy);
            vy = checkBottomCollision(vy);
        }
    }
}
```

```
public void run() {
    setUp();
    addMouseListeners();
    while(NTURNS > 0)
    {
        while(true){
            vx = rgen.nextDouble(1.0,3.0);
            if(rgen.nextBoolean(0.5)) vx -= vx;
            vy = 0.1;
            moveBall();
            vx = checkVerticalCollision(vx);
            vy = checkTopCollision(vy);
            vy = checkBottomCollision(vy);
        }
    }
}
```

FIGURE 1 Two consecutive snapshots of student code in which three lines were added. In this case, the code update differential is (3, 0, 0, 71, 0, 0): 3 lines added, 0 lines deleted, 0 lines modified, 71 characters added, 0 characters deleted, 0 characters modified.

choice for this metric is based on the fact that these data, in addition to being relatively unproblematic to extract from our data set, could be strongly related to the ways in which students plan and execute their work while programming, and thus would be a good compromise between low computational cost and usefulness.

In Figure 1 we see an example of two consecutive snapshots of student code in which three lines of code were added but nothing else was deleted or modified.

We focus on the *size* and *frequency* of code updates. As students add and remove lines of code, they can do so in large chunks or small pieces (size) and either very frequently or in just a few episodes (frequency). The combination of *size* and *frequency* is what we call the *code update pattern.*

In most of our analysis we assume that small, frequent code changes could represent episodes of *tinkering,* whereas larger, less frequent code changes could represent episodes of *planning.* We sometimes assume that planning would be the more advanced and desired behavior, as the prompts that students received for all assignments contained consistent advice about planning and programming methodology. We are not implying that it is, universally, the best possible behavior (experts could tinker as well, but very few experts were enrolled in this introductory class), however it was the behavior emphasized in this class and in the grading of the assignments. This assumption is important for when we compare the code snapshot data with course grades, but we acknowledge the limitations of this assumption when necessary.

balance between detecting modifications and not having any modifications. When we looked at the data, using values closer to 40% resulted in the inclusion of lines that did not appear to truly be modifications (low precision). In contrast, using 20% resulted in poor recall, such that many modifications were overlooked. Hence, we chose a similarity score of 30% as the criterion.

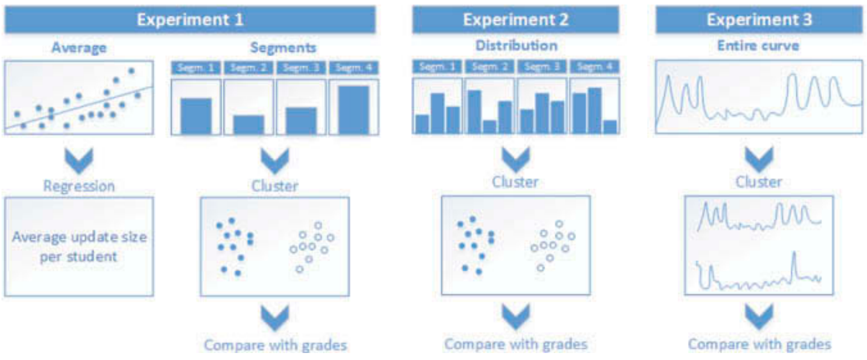


FIGURE 2 Summary of the three experiments in our preliminary explorations: We begin with one data point per assignment ("Average"), then four ("Segments"), 12 ("Distribution"), and 30–100 ("Entire curve") in Experiment 3. Segm = segment.

In Figure 2 we have a summary of the experiments and methods. In Experiment 1, we first try to relate the average update size for each student and his or her grades using a regression (one data point per assignment). Next, instead of using the average update size for each assignment, we break each assignment into four segments and start using clustering techniques (four data points per assignment). In our second experiment, we further calculate the per-segment distribution of code update sizes using three bins (large, medium, and small) and again use clustering techniques (12 data points per assignment). Finally, in Experiment 3, we use the entire code update pattern instead of just the binned sizes of the updates (30–100 data points per assignment). We generate code update curves for each student for each of the four assignments, use the similarity between those four curves to calculate a general score for how much each student's pattern changed during the course, and try to find correlations between the amount of change and course grades.

In summary, we start with the aggregate data, then segment them in time, then resegment them in terms of the distribution, and finally use all of the individual data points without any aggregation. Consequently, the dimensionality of the data increases, and we accordingly move from traditional statistical methods (regression), to simple machine-learning algorithms (clustering), and finally to similarity metrics.

## Data Collection

The data for this analysis were derived from four programming assignments completed over several weeks of instruction by two cohorts of students in two academic quarters (spring and fall of 2012). The spring data consisted of assignments completed by 74 students and included 14,000 code snapshots in total. The fall data consisted of about 140,000 code snapshots generated by 272 students.

All students were undergraduates or graduate students enrolled in a programming methodology course at a research university. Class lectures, which met three times a week, were supplemented by weekly discussion sections with teaching assistants. Students completed seven assignments during a 10-week period (we refer to them as *Assignment 0, Assignment 1,* and so on). Out of the seven assignments, three were not used: Assignment 0 was done in a different programming language, Assignment 5 was only a series of quick programming exercises and not a proper programming project, and Assignment 6 had excessive missing data due to technical problems in the data capture (the remaining assignments are described in detail in the supplemental material).

## Experiments 1 and 2: Does the Average Size of the Code Updates Correlate With Course Performance?

Our first hypothesis about possible patterns in the data was that advanced or higher performing students would plan more, and thus write larger chunks of code at a time, and that novices would tinker more, and therefore make small updates to the code frequently. Our goal for the first experiment was to answer whether *the average size of the code updates correlates with course grades.*

The simplest way to go about answering this question was to do a simple regression between exam grades and average size of the code updates per student. However, the regression showed no significant results, $F(1, 279) = 0.005$, $p < .94$. We hypothesized that this could have been an artifact of the averaging: Students could start an assignment making small updates and end it making large ones, but the averaging would erase these changes. We then sliced the data into more segments to mitigate this effect, dividing each assignment into four equal time-based segments, and extracted the average of each segment. With four data points per assignment per student, we turned to basic clustering techniques from machine learning. Clustering algorithms are used to segment multidimensional data points into distinct groups: They iterate several times through the data, calculating how different distinct clusters would be given initial parameters, and pick the parameters that maximize the difference between the groups (there are many types of clustering techniques: k-means, x-means, and many others; see the supplemental material). With these more detailed data, we used the x-means clustering algorithm[2] to see whether the resulting clusters would attain different average grades in the course. Even though the clusters appeared to be significantly different from each other, the results with regard to grades showed effect sizes on

[2]We attempted to use k-means as well, and the algorithms converged to the same clusters.

the order of 0.2, which is quite small (see the supplemental material for a full explanation of this algorithm), so we were still unable to make claims about the impact of the code update size on course performance. After these weak results, we considered that the averaging of the code update size per segment was still occluding differences among students. Perhaps the distribution of update sizes was more informative than the average—for example, some students could have a bimodal update size distribution, whereas others would have a predominant size. Thus, we used the distribution of update sizes for each segment (we calculated the proportion of large, medium, and small changes, so we had 12 data points per assignment). In this second analysis, the x-means algorithm again produced two quite distinct clusters, but the difference in average final grades corresponded to a Cohen's *d* of 0.21, which is still quite small. In these two experiments, we tried to use increasingly high levels of detail about the data, and more sophisticated statistical techniques, to validate our hypothesis, and results suggested that in each experiment we got an increasingly strong distinction between the two clusters but no predictive power in terms of course performance. Thus, it seems that the answer to our first research question (Does the amount of tinkering, as measured by the average size of the code updates, correlate with course performance?) was negative.

## Experiment 3: Does the Amount of Change in Students' Programming Patterns Predict Course Performance?

Given the relatively small effect size of the code update size on course performance from Experiments 1 and 2, in Experiment 3, instead of focusing on the size of the updates, we attempted to use a more sophisticated measure of students' programming patterns. Instead of just using the differences in *size,* we also included the differences in *frequency*—we call the combination of size and frequency the *code update pattern.* Imagine that, for Assignment 1, a student compiled or saved her code 30 times, adding on average 25 lines of code each time. If this student started with large updates and moved toward small updates, her curve (see Figure 3) would have many high peaks corresponding to the first (and large) code updates and then many small peaks at the end corresponding to small updates.

Now imagine that the same student, in the next assignment, had only large code updates—in this case, we would see a curve with very high peaks and no small ones. If we assume that the assignment prompts were similar, we could hypothesize that this student changed her coding behavior—the behavior of making small code updates in the second half of the first assignment disappeared in the second assignment. Thus, if we had a way to compare those two curves, and see how different they are, we could possibly detect that a student changed her coding behavior based on the calculation of the difference between the two curves. Fortunately, there are simple techniques in machine learning that allow us

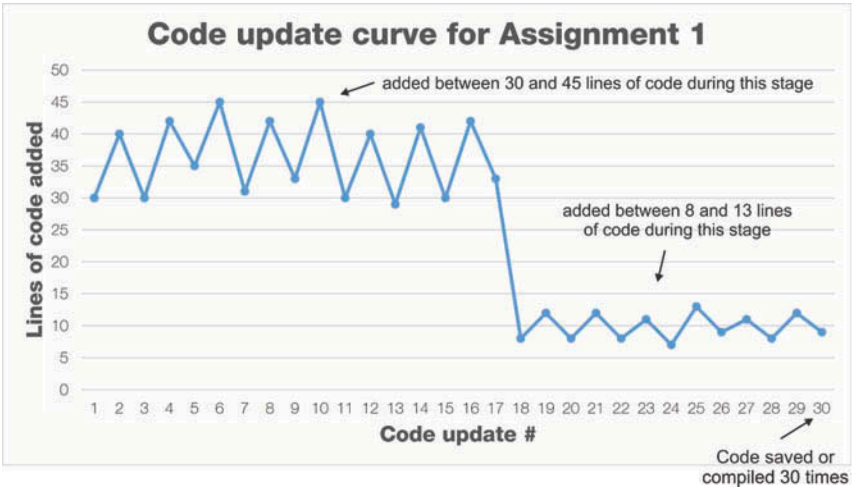


FIGURE 3 Hypothetical example of a code update curve, showing large code updates at first and smaller ones toward the end of the assignment.

to compare complex data sets even if they have different sizes, so we make use of them in this analysis.

In this experiment, we generated curves for all students in four assignments (1–4) and then calculated—for the *same* student—how different his or her curve in Assignment 4 was from his or her curve in Assignment 3, then Assignments 3 and 2, and so on. We hypothesized that if a student had a very similar curve in both Assignments 4 (the last) and 1 (the first), there would be very little change in his or her code update pattern. Conversely, if the curve was wildly different, this student was considered to have changed more his or her update pattern from the initial to final assignments.

The research question in this experiment was, *Is there a correlation between how much those curves change and students' performance as measured by grades?* We believe that it is reasonable to expect that the changes in these curves would be correlated with course performance for the following reasons:

1. The course was named Programming Methodology (thus it teaches not just programming concepts or techniques but methods), and a strong emphasis was given to learning systematic approaches to programming, such as decomposition and planning.
2. It was expected that one of the learning outcomes of the course would be that students would change the way they approached programming.

3. Considering that this was an introductory course, most students were novices; very few experts took the course (arguably experts could have different types of behaviors or come in with their own expert ways and patterns of updating code).

Therefore, the hypothesized scenario would be that lower performing students should exhibit little change in their update patterns, and higher performing students would show a great deal of change in their update patterns. In what follows, we go deeper into the methodological aspects of the experiment and then comment on the results.

*Data Standardization, Normalization, and Metrics.* After we extracted all of the data from the code snapshots and calculated the code update differentials, we normalized the size of every code change relative to the class average and standard deviation.[3] Next we used two machine-learning methods to calculate the difference between two given curves. First, we used a technique known as dynamic time warping to stretch the curves in order to calculate the differences between them when they had different sizes. This addressed the problem of sequences of different lengths, as students might have produced a different number of snapshots per assignment. Second, we used a technique known as scaled dynamic time warping distance to calculate the actual difference (or *distance,* in machine-learning terminology) between two curves (more details about both of these methodologies are given in the supplemental material).

*Data Analysis and Categorization of Students.* The determination of the difference between students was done by comparing later assignments with earlier ones. We observed whether each student's update curve for Assignment 4 was most similar to that for Assignment 1, 2, or 3 and how Assignment 3's curve compared to those for Assignments 1 and 2. Thus, we performed five pairwise similarity comparisons for each student—three for Assignment 4 (4 and 3, 4 and 2, 4 and 1) and two for Assignment 3 (3 and 2, 3 and 1)—and chose the two pairs that were the most different, as indicated in Table 1 .

Each student therefore could have either two, one, or zero high-change pairs. It is crucial to understand what a high- or low-change pair means for the rest of our analysis. If the last assignment in the course (4) was done in a similar fashion as the first (1), we consider that there was not much change (thus 4 and 1 is a low-change pair). Based on the simple count of high- and low-change pairs we assigned a change score to each student and grouped students into six groups

[3]The normalization was meant to avoid the fact that absolute values of the changes skew the analysis or erase assignment-specific characteristics that we would want to account for (instead of attributing to individual differences).

TABLE 1
List of All Pairwise Similarity Comparisons

| *Possible Pairwise Comparisons* | *If the Most Equal Pair Is . . .* | *This Indicates . . .* |
|---|---|---|
| 4 and 3 \| 4 and 2 \| 4 and 1 | 4 and 1 | Low level of overall change |
| 4 and 3 \| 4 and 2 \| 4 and 1 | 4 and 2 | Medium level of overall change |
| 4 and 3 \| 4 and 2 \| 4 and 1 | 4 and 3 | High level of overall change |
| 3 and 2 \| 3 and 1 | 3 and 1 | Low level of overall change |
| 3 and 2 \| 3 and 1 | 3 and 2 | High level of overall change |

TABLE 2
Group Proportions

| *Group Label* | *Percentage of Class* |
|---|---|
| A (highest change) | 8 |
| B | 8 |
| C | 22 |
| D | 12 |
| E | 15 |
| F (lowest change) | 35 |

based on it (A, B, C, D, E, F). Group A was composed of students with the highest degree of change from Assignments 1 to 4, and Group F was composed of students with the lowest. Students in Group A had *two* of the high-change pairs (see Table 1); those in Groups B, C, and D had just *one* high-change pair; and those in Groups E and F had *none*. Table 2 shows the proportion of students in each group for our data set after these calculations.

*Results: Students' Progression.* We started by ranking the A–F groups in terms of their assignment, midterm, and exam grades and then checked whether the groups that were supposed to perform better actually had a higher grade. Table 3 shows the rankings for all students in terms of their assignment grades. The reader should read the table in the following way: On Assignment 2, students in Group C had an average grade higher than that of any other group in the class; on Assignment 4, the average grade for students in Group E was the lowest in the class; and so on. We would expect to find Groups A and B on top of most of the rankings, but there were some surprises. Group B was the lowest performing group on two assignments, and Group A was never the top performing group. Those surprises were present in subsequent rankings as well. However, the reader must remember our caveat from the beginning of this study: Our goal is to examine whether our measures have any correspondence to real-world performance

TABLE 3
Rankings for Five Assignments, Showing the Relative Performance for Each Group, and the Count of High-Change Pairs

| *Rank* | *Assignment 0* | *Assignment 1* | *Assignment 2* | *Assignment 3* | *Assignment 4* | *Count of High-Change Pairs* — *6 Groups* | *Count of High-Change Pairs* — *3 Groups* |
|---|---|---|---|---|---|---|---|
| 1 | D ▪ | C ▪ | C ▪ | C ▪ | C ▪ | ▪▪▪▪▪ 5 | ▪▪▪▪▪▪▪▪▪▪▪▪ 12 |
| 2 | C ▪ | E | A ▪▪ | A ▪▪ | A ▪▪ | ▪▪▪▪▪▪▪ 7 | |
| 3 | A ▪▪ | A ▪▪ | F | F | D ▪ | ▪▪▪▪▪ 5 | ▪▪▪▪▪▪▪▪▪ 9 |
| 4 | F | D ▪ | D ▪ | B ▪ | B ▪ | ▪▪▪▪ 4 | |
| 5 | B ▪ | F | E | E | F | ▪ 1 | ▪▪▪▪ 4 |
| 6 | E | B ▪ | B ▪ | D ▪ | E | ▪▪▪ 3 | |

*Note.* We use the symbol ▪ to indicate the number of high-change pairs in each group.

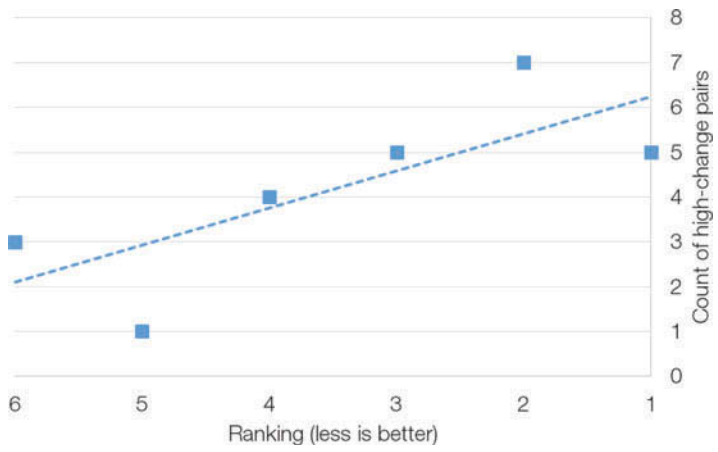


FIGURE 4 Amount of change in pattern versus grades on assignments.

measures not to build a predictive model or test a statistical hypothesis. Because we are doing data mining in a very unstructured, noisy, and complex data set, we are concerned with general trends and patterns. To better identify these trends, we further aggregated the rankings by counting the number of high-change pairs across each rank. As we move up Table 3 (thus looking at groups with better performance), the high-change groups are clearly more dominant, which is even more apparent when we use only three groups (see the last column of Table 3). This confirms our expectation that these groups (A, B, C, D) would also, on average, perform better in the course than Groups E and F. The values for the count of high-change groups are plotted in Figure 4, and they were highly correlated despite only trending ($R^2 = .577$), $F(1, 4) = 5.45$, $p < .08$.

TABLE 4
Rankings for the Final and Midterm Exams, Showing Which Group Performed Better, and the Count of High-Change Pairs

| | | | *Count of High-Change Pairs* | |
|---|---|---|---|---|
| *Rank* | *Midterm* | *Final* | *6 Groups* | *3 Groups* |
| 1 | A ▪▪ | A ▪▪ | ▪▪▪▪ 4 | ▪▪▪▪▪▪ 6 |
| 2 | C ▪ | C ▪ | ▪▪ 2 | |
| 3 | D ▪ | F | ▪ 1 | ▪▪ 2 |
| 4 | F | D ▪ | ▪ 1 | |
| 5 | E | E | 0 | ▪▪ 2 |
| 6 | B ▪ | B ▪ | ▪▪ 2 | |

*Note.* We use the symbol ▪ to indicate the number of high-change pairs in each group.

When we look at exam and midterm grades we find a similar type of result, although less statistically significant. Group A did outperform all others, but still there were some anomalies (e.g., Group F was among the top performers on the midterm and final examinations). Despite these surprises, a similar aggregate trend was observed. In Table 4 we present the same analysis for the midterm and final exam. Even though the data showed a strong correlation, it was not significant, so we can only characterize it as a trend ($R^2 = .392$), $F(1, 4) = 2.58$, $p < .18$. However, a visual inspection of the plot (see Figure 5) shows that the point corresponding to rank = 6 could be an outlier; given that we are concerned with general trends, if we exclude this point, we obtain an extremely strong fit ($R^2 = .938$), $F(1, 4) = 22.09$, $p < .018$. Also, examining the last column of Table 4, we see that when we again break down the ranking into only three groups, the trend of the top groups to perform better is quite apparent (6 for Ranks 1 and 2 vs. 2 for the others).

**Further verification of ranking trends.** These rankings for exams and assignments were created by comparing the average grades of a group with those of all other groups. One possible concern with this analysis is that many of the *p* values computed from the Student's *t* tests were close to .05 and that we undertook several different comparisons between groups. Some researchers use post hoc analysis (e.g., Benjamini and Hochberg) to control for false discovery rate. However, false discovery rate detection tends to increase the probability of Type II error, which is the probability of missing a true discovery. Another concern is that we undertook several pairwise comparisons for which not all differences exhibited statistical significance at the .05 level. We could address both of these concerns by looking at the four assignments and two examinations and computing

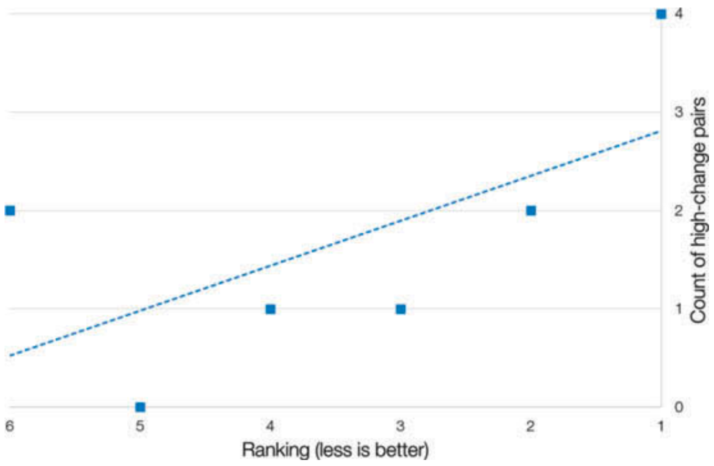


FIGURE 5 Amount of change in pattern versus grades on midterm and final exam.

TABLE 5
Group Rank Probabilities Based on Assignment and Examination Ranks

| *Groups* | *Probability of Ranking* |
|---|---|
| A | 0.008 |
| B | 0.008 |
| C | 0.016 |
| D | 0.250 |
| E | 0.006 |
| F | 0.008 |

the likelihood of each group receiving a given rank, or a very similar rank, across the different metrics. For example, in this experiment, we found that Group B ranked 6th on both the final and the midterm examination. Although it is possible that this ranking was random, the probability of this happening (Group B receiving Rank 6 on both exams) is only 2.8%. Table 5 reports the probability of each cluster's ranking for the examination and assignment scores, and the probability of randomly arriving at the cluster rankings revealed by the data was extremely low for nearly all of the clusters that we identified. For all groups except for C and D, the probability of randomly attaining the assigned rankings was close to or less than .008 (which corresponds to a .05 significance level after Bonferroni adjustment).

## Discussion

In our three experiments, we intended to develop initial methodologies to conduct fully automated measures of programming activity, determining promising directions and possible dead ends as we tried to answer the following research question: Can relatively simple quantitative measures capture differences in students' behaviors when programming, and are those differences related to course performance?

In the first two experiments, we tried to use the size of the code updates as an indication of programming style and relate it to course performance. Both experiments revealed that although well-defined clusters could be determined, showing that in fact there could be real generalizable programming styles within the cohort, these did not correlate strongly with performance in the course, despite our attempts to slice the data in different ways. If we believe that large code updates are related to more planning (which we find reasonable to assume given our previous exposition), this last finding could be counterintuitive for the computer science education community: Perhaps it is not as determinant to success as believed.

That led us to the third experiment, in which we used not only the code update sizes but the code update pattern, or the curve describing the time series of all code update sizes. We also changed the approach by not measuring an absolute per-assignment metric but looking at the amount of *change* in students' update patterns. Thus, the main component of Experiment 3 was the grouping of students on the basis of the extent to which their programming patterns changed over the course of the class. We initially hypothesized that students who exhibited high change (abandoning old update patterns and developing new ones) would also be high performers. The rationale was that students entering the class were novices, and they would supposedly learn programming methodology and steadily adopt the behaviors taught in class, which were primarily related to more planning and intentional decomposition.

The results indicated that there was a connection between students' grades and the amount of change in their programming patterns, confirming our hypothesis that students with higher grades would also change their programming patterns the most. We acknowledge that the results are still not statistically strong (in the first analysis, our best $p$ value was .08), but our post hoc analysis was coherent with these results for four of the six groups at the $p < .05$ level. In any case, given that our goal was to provide early indication of the promise and limitations of these techniques, this level of significance is adequate for the task at hand.

The correlations we found are just a first step in the direction of developing measures for programming patterns such as tinkering and planning, or how much students change their patterns, and the results seem to agree with the early literature in this nascent field, which suggests a relationship between the ways in which students update their code and their programming experience (Berland & Martin, 2011; Blikstein, 2011b). It seems that the answer to our research question is that simple, unsupervised quantitative measures in learning analytics might be useful for determining the existence of a pattern in the data but not necessarily the relationship between a given pattern and students' behavior. Given these limitations, in Study 2 we turned to more complex methods to try to draw a connection between machine learning–induced patterns, students' programming patterns, and students' course performance.

## STUDY 2: STUDENT PATHS IN A SINGLE ASSIGNMENT

Whereas in the three previous experiments we showed students' progress over several assignments, the goal of Study 2 was to examine students' progression within a single assignment, exploring in detail their trajectories. We chose the first assignments in the course, Checkerboard Karel, in which the goal is to program a robot to produce a particular pattern in two-dimensional graphical windows (see Figure 6) of different sizes, placing beepers (the grey diamond-shaped icons) in each of the windows. This setting utilizes an implementation of the Karel the Robot programming language (Pattis, 1981), which uses a much circumscribed portion of the Java programming language (Piech et al., 2012). The implementation includes commands such as "turnLeft," "move," and "putBeeper." An example of a Karel program follows, in which the robot is made to fill empty spaces with beepers:

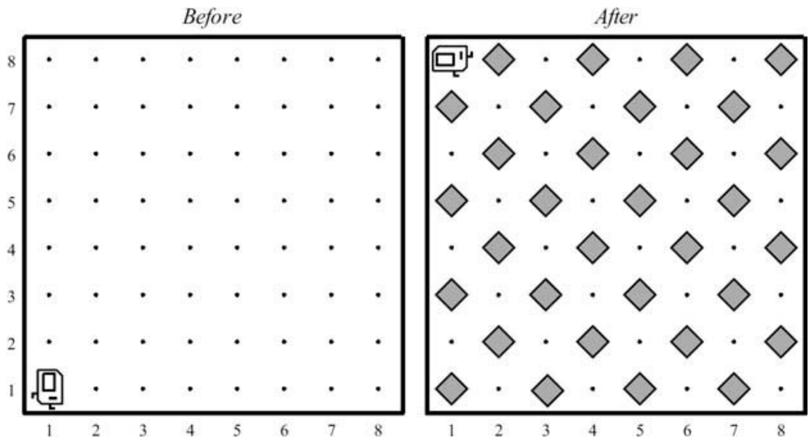


FIGURE 6 The Checkerboard Karel assignment: Students must program the robot to create the pattern on the right, using a simplified programming language.

```
private void Example() {
turnRight();
move();
if (noBeepersPresent()) {
putBeeper();
}
turnAround();
move();
turnRight();
}
```

This assignment is ideal for a deeper machine-learning analysis because Karel hides the complexities of real Java programming through the provision of a simpler command set. Over the years, the instructors for this course, two of whom are coauthors, have observed that a considerable number of students are able to submit working and successful Karel assignments without any prior command of basic programming concepts, as revealed by tests and tutoring sessions with teaching assistants. Consequently, assessing students' final submissions was demonstrated to be insufficient for gauging students' understanding of programming—thus this data set would provide a good opportunity to investigate what takes place throughout the entire process.

## Data Collection

The Checkerboard Karel assignment allows students to gain experience with program flow and decomposition. A full solution must work on Karel worlds of any size and geometry. We collected 370 Karel assignments (238 from Spring 2011 and 132 from Summer 2011[4]), with an average of 159 code snapshots for each student ($SD = 82$). Each snapshot was time-stamped and could be run through a simulator to record errors and to test functionality.

## Data Analysis

*Program Distance Metric.* Whereas in the previous section we only utilized the size of the code updates without examining the code itself, in this analysis we dive into the content of the program updates. To model students' progress through an assignment, it was necessary to determine whether there were common intermediate states across the programs of many students. The first step was to construct a method to compare two assignments (one from Student A, one from

[4]The teaching team varied between the spring and summer quarters, but the curriculum remained unchanged.

Student B) against each other to determine whether they contained similar intermediate states. We considered three algorithms for calculating similarity (more details about these algorithms are in the supplemental material):

1. Bag of Words Difference: We built histograms of the different keywords used in a program and used the Euclidean distance between two histograms as a naïve measure of the similarity (Salton, Wong, & Yang, 1975).
2. Application Program Interface (API) Call Similarity: We ran each program with typical inputs and recorded the resulting sequence of commands and functions executed. We used the Needleman–Wunsch[5] algorithm for global DNA alignment (Needleman & Wunsch, 1970) to measure the difference between the lists of commands and functions executed (or API calls) by the two programs. API calls are a particularly good representation of Karel programs; because they do not store variables, the entire functionality of a program can be expressed in terms of the usage of the language's commands (e.g., turnLeft, move). This metric is more difficult to implement for full-blown programming languages such as Java.
3. Abstract Syntax Tree (AST) Change Severity: An AST is a systematic approach to breaking down a computer program into its constituent elements while capturing the relationships between them. It is essentially a graphical hierarchical representation of a program that allows for comparison across different programs. We built AST representations of both programs and calculated the minimum number of rotations, insertions, and deletions needed to transform the AST of one program into the AST of another program, using the Evolizer algorithm (Gall, Fluri, & Pinzger, 2009).

All of our similarity metrics were normalized by the sum of the distances between the programs and the starter code (the small amount of code students were given initially by the instructors). To evaluate each metric we had a group of five advanced computer science students with teaching experience (whom we subsequently refer to as *experts*) label sets of random snapshot pairs as either similar or different. All experts looked at different snapshots, meaning that there was no overlap in expert ratings through which to compute interrater reliability; instead, we based our results on the assumption that the error in the labeling would average out across a large number of comparisons. This labeled data set was used to measure how well the distance metrics could replicate the expert labels. The experts were asked to assess similarity based on a rubric that had them identify major and minor stylistic and functional differences between pairs of programs.

[5]See the supplemental material for details.

We selected a sample of 90 pairs of programs, capturing a balanced spectrum of similar, dissimilar, and slightly similar code. Compared to the human labels, the API Call Similarity performed best, with an accuracy of 86% ($p < .00001$) versus 55% for Bag of Words and 75% for AST Change.[6]

The distance metric used to build our model was a weighted sum of the AST Change metric and a set of API similarity scores (each generated by running the programs with a different input world). We trained a support vector machine[7] with a linear kernel (Shawe-Taylor & Cristianini, 2000) to come up with the composition of our different distance measures that was best able to reproduce the human labels. The values used to weight the distance measures in our composite similarity score were the weights assigned to each distance measure by the support vector machine.

*Modeling Progress.* To understand how we modeled students' progress, consider a hypothetical example (see Figure 7). A student begins with the starter code given by the instructors (A) and writes code to instruct Karel the Robot to place a single beeper in the world (B). Next she writes code to create a single row (C), then expands the code to create two rows (D), and finally comes up with a solution that works on worlds of all sizes (E). This trajectory has states ("program has the capability places a single beeper," "program can place a perfect row in all worlds," etc.) and transitions between these states; this set of states and transitions is called a *finite state machine* (a more detailed definition of finite state machines is given in the next section).

We can use a similar procedure to determine whether there are patterns in the way in which different groups of students move from one state to another. In Figure 7 we picture a simple linear sequence, but there are many ways a student can arrive at the solution. Learning the sequence of states that students take as they solve the assignment reveals both productive and unproductive paths, and our main goal was to find a method for automatically categorizing students'

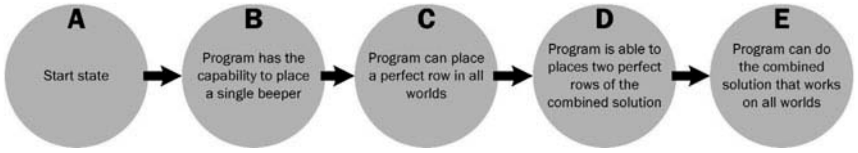

A
Start state
B
Program has the capability to place a single beeper
C
Program can place a perfect row in all worlds
D
Program is able to places two perfect rows of the combined solution
E
Program can do the combined solution that works on all worlds

FIGURE 7 A sequence of states that a student could generate.

[6]Because the similarity metric could be biased toward assigning low similarity scores to snapshots of assignments by the same student, we modified our algorithms to never use the similarity value computed from the same student.

[7]A support vector machine is a sorting machine that learns by example. See the supplemental material for details.

development paths into clusters and verify whether course performance was correlated with the clusters. In order to develop this method, we had to address several issues. First, we needed to identify the most typical states that students could occupy at a given time. This entailed clustering student snapshots by similarity. Second, we had to find the typical paths through these states and then find similarities in these paths.

*Technical Implementation.* Because this section is quite technical, some readers might want to skip it. We are aware that much of this detail might not be relevant, but it is necessary to demonstrate the soundness of the results.

Hidden Markov model (HMM). The first step in our student modeling process was for the system to learn (in a machine-learning sense) a high-level representation of how each student progressed through the assignment. To enable this process, we modeled a student's trajectories as an HMM (Rabiner & Juang, 1986). In general terms, a Markov model is a sequence-based model for which the likelihood of being in a particular state at a given time step is dependent only on the state at the previous time step. In a *hidden* Markov model the probability of being in any given state must be inferred from observations, because the state itself is a hidden (latent) variable. As an example, consider a person walking through a dark house. At any point, the person does not know what actual location (state) in the house she occupies, yet she is able to make some observations based on the walls and doors she can feel (observe) at her current location. Such observations allow that person to infer her true location (state) on the basis of these observations as well as on the basis of what she believes her previous location in the house (state in the prior time step) to have been.

The HMM we used (see Figure 8) proposes that for each snapshot, a student is at a milestone corresponding to one of the defined programming *states.* Although we cannot directly observe the state (it is a latent variable), we can observe the code snapshots, which are approximate indicators of the state. Example states might include "the student has just started to program," "the student's program can place a single beeper," or "the student's program can generate two perfect rows." Note that these states do not need to be explicitly labeled in the model. Given the student trace data provided, they are autonomously induced by the algorithm.

There are two key parameters in the HMM: the probability of a student going from one state to another and the probability that a given snapshot actually follows from a particular milestone. Modeling students' progress in this way makes a simplifying assumption: Given the present state, the future state of a student's code is independent of the past states. This assumption is not entirely correct, as students still remember and learn from past experiences within a given assignment. Nevertheless, this simplification maintains algorithmic tractability while remaining useful for finding patterns.

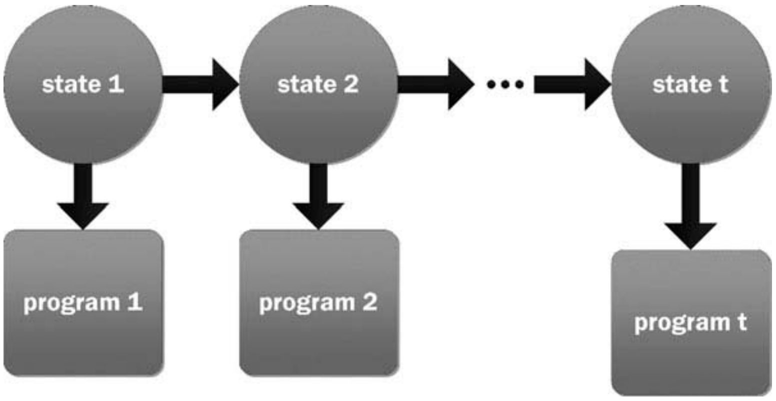


FIGURE 8 The program's hidden Markov model of state transitions for a given student. The node "program *t*" denotes the code snapshot of the student at time *t,* and the node "state *t*" denotes the high-level milestone that the student is in at time *t.*

Dynamic time warping. We captured snapshots every time students compiled or saved their code, but this is not an accurate representation of a unit of work. Different students might exhibit different types of behaviors regarding how often they save or compile their programs. However, a feature of our approach is that it assumes a relatively constant rate of change between observations, so we again used *dynamic time warping* to compare the nonsynchronized data from different students.

Understanding the next steps in this model requires a definition of a finite state machine and an explanation of how it works. A finite state machine is a directed graph containing a finite set of states (the nodes in the graph) along with a finite set of transitions between those states (the arrows in the graph). The transitions indicate how a process can move from one state to another over time. For example, a finite state machine modeling how students transition between different courses in mathematics might begin with prealgebra and have other states for algebra, trigonometry, precalculus, and calculus. Although many students may take a linear path through these subjects, some students might skip a class (e.g., transition from trigonometry directly to calculus). Thus, each state from our HMM becomes a node in the finite state machine, and the weight of each arrow between nodes provides the probability of making the transition from one state to the next.

In order to create the HMM we had to first identify three variables:

- The set of discrete high-level states a student could be in (as defined by the set of code snapshots belonging to that same state).

- The probability of being in a state given the state that the student was in previously (called the *transition probability*).
- The probability of seeing a specific code snapshot given that the student is in a particular state, called the *emission probability.* To calculate the emission probability, we interpreted each of the states as emitting snapshots with normally distributed similarities.

To predict these variables in the HMM, we used the expectation maximization algorithm (Dempster, Laird, & Rubin, 1977). To compute the different milestones of the assignment, we used two clustering algorithms: k-medioid (Kaufman & Rousseeuw, 1990) and hierarchical agglomerative clustering (Cutting, Karger, Pedersen, & Tukey, 1992). Once we had established the set of possible states (i.e., milestones) for the assignment, we used an expectation maximization algorithm to simultaneously compute both the transition and emission probabilities in the state diagram, which resulted in a probabilistic assignment to the state variables for each student at each point in time as well as estimates for parameters for the HMM reflecting the state diagram induced.

*Finding Patterns in Paths.* The final step for our algorithm was to discover patterns in the students' transitions through the different states. To find these patterns, we clustered the paths that students took through the HMM using a method developed by Smyth (1997). For all students, HMMs were constructed to represent their individual state transitions, incorporating prior probabilities based on the HMM for the entire class. We measured similarity between two students as the probability that Student A's trajectory could be generated by Student B's HMM and vice versa. As we explain in the next section, we then clustered all of the student paths to create groupings of students based on the characteristics of their paths.

## Results

*Defining the Clusters for Students' Pathways.* The clustering of a sample of 2,000 random snapshots from the training set returned a group of well-defined clusters (see Figure 9). One important step is to define the number of possible clusters. We based our number on the silhouette score of the clusters' goodness of fit. This score indicates the best number of clusters, and the number that maximized the silhouette score was 26. A visual inspection of these clusters confirmed that snapshots that clustered together were functionally similar pieces of code and showed an ample range of the different code snapshots that we had observed. Our results were analogous when we tried a slightly different number of clusters. When we repeated this process with a different set of random snapshots, 25 of the

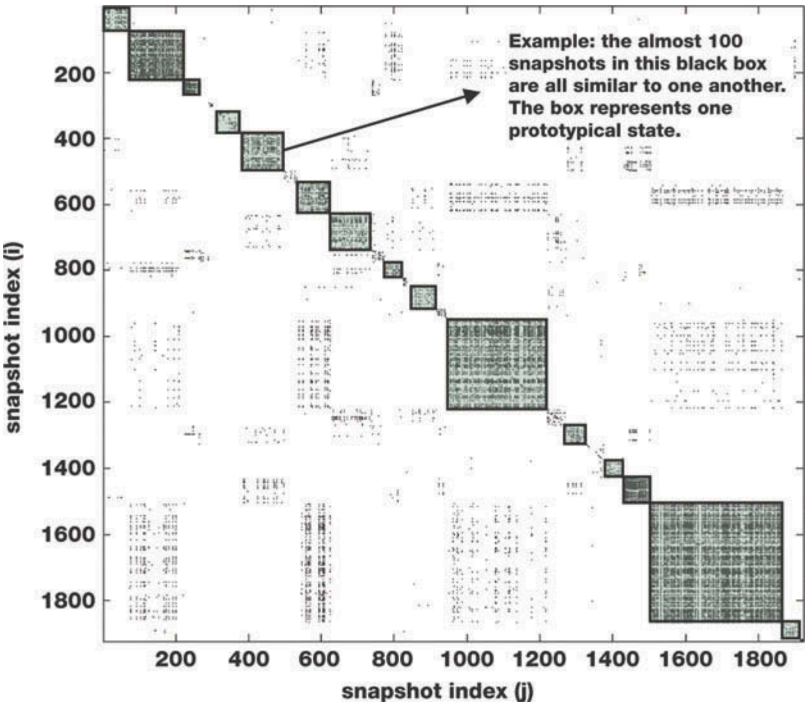


FIGURE 9 Dissimilarity matrix for a clustering of 2,000 snapshots. Each row and column in the matrix represents a snapshot (snapshots are sorted so that more similar snapshots are in adjacent rows). The entry at row i, column j, represents the similarity of snapshots i and j (darker indicates greater similarity). The 26 prototypical states are visible as blocks along the main diagonal—we highlighted the larger blocks with thick lines around them, and made them darker. The less visible blocks, in light gray and outside of the main diagonal, represent snapshots that could not be grouped in any of the 26 states.

27 clusters were identical, indicating that the results were quite stable—the choice of 26 clusters and the high-level trends were robust.

*Sink States.* The state machine for the entire class showed that there were several sink states—milestones where the students had clearly encountered serious difficulties. We interpret these states as programs with bugs that were very hard to solve. Once a student entered into such a state, he or she had a high probability of remaining there for several code updates. The state machine indicated the probability and which transition most students took to get out of that state. For example, some students would get their code into a state in which the program would enter an infinite loop and keep placing checkers only in the first two rows.

Once in this sink state, students tended not to make forward progress for many cycles of saving or compiling.

*Prototypical Patterns and Relationship With Class Midterms.* Utilizing the 26 possible states and the clustering of trajectories, we were able to generate a graphical description of those trajectories in Figure 10 (the size of the circles is proportional to the number of code snapshots in a given state). We can observe clear differences in both the types of states that the two groups visited and the patterns that characterized the students' transitions between states. Qualitatively speaking, the Gamma group can be described as becoming trapped in several sink states (e.g., L and K) and then making a direct transition from a semi-working program (state I) to a fully functional program (state E). The Alpha group seemed to make smaller but steadier incremental steps toward the solution.[8]

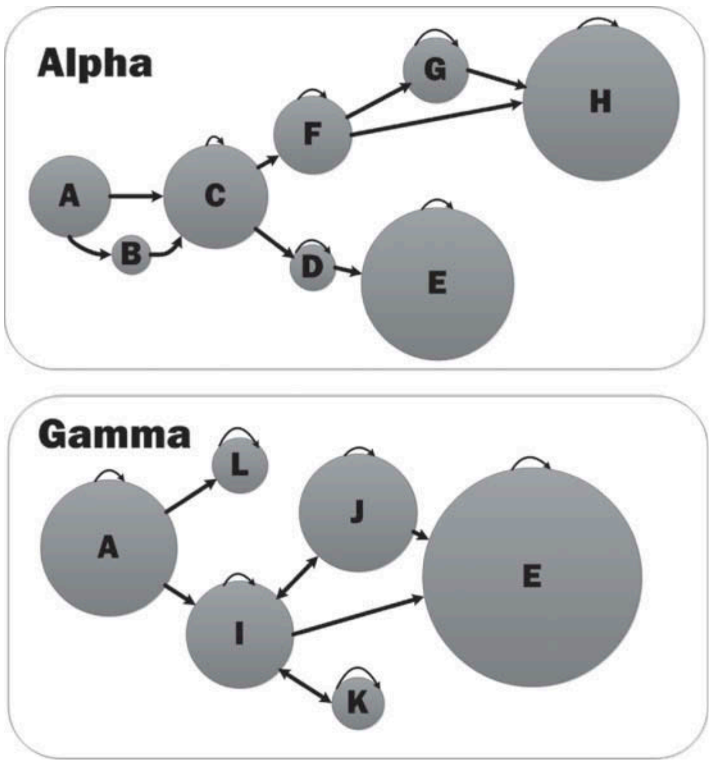
*Correlation With Midterm Grades.* In addition to finding patterns in students' development progress, which can be indicative of a student's programming style and knowledge of programming concepts, we also wanted to determine the predictive power of such algorithms for students' performance in exams. Thus, we sought to examine the extent to which students' development trajectories on their first assignment could be used to determine their performance on the midterm exam.

After defining the Alpha, Beta, and Gamma clusters (see Table 6), we found that the group a student was clustered into was indeed predictive of his or her midterm grade. The distinction between the Alpha and Gamma groups was particularly large, having a mean midterm score difference of 7.9% ($p < .04$, two-tailed $t$ test).

*Results: Comparison With Other Cohorts.* In order to understand the generalizability of such models outside of our sample, we also classified learners who took the class during the following quarter into the Alpha, Beta, and Gamma groups using the training set from students from the previous quarter. The Alpha group ($n = 30$) had a mean midterm score of 69.7 ($SD = 14.1$), and the Gamma group ($n = 42$) had a lower midterm score of 63.3 ($SD = 15.4$). This time we found a trending difference between the mean midterm scores of these groups ($p < .08$).

This last result shows that the induced models are quite robust: They capture patterns in development paths that generalize across student development behavior between classes with different instructors and are not specific to a single class. The generalizability of the induced models has the potential to provide predictive capability in helping address student difficulties in programming and,

[8]For space considerations and because the data were more robust and well-defined, we only show Alpha and Gamma.



| State | Functionality of the typical snapshot. In this state, the program… |
|---|---|
| A | …is beginning (start state) |
| B | …can place a single beeper |
| C | …can place a perfect row in all worlds |
| D | …can place two perfect rows of the combined solution |
| E | …has the combined solution that works on all worlds |
| F | …has the zigzag solution that works in all worlds except one-column worlds |
| G | …has the zigzag solution that crashes on add column worlds |
| H | …has the zigzag solution that works on all worlds |
| I | …can place one line, misses a beeper on odd-column worlds |
| J | …can do two rows of the zigzag solution; Second row shifted |
| K | …repeats first two rows indefinitely |
| L | …is hardcoded to place first row in the $8 \times 8$ world |

FIGURE 10 Visualization of finite state machines for the Alpha and Gamma clusters of students. The size of the circles is proportional to the approximate number of code snapshots in the various states. Note that this is not equivalent to the number of students in each state, as the same student could have multiple snapshots within any given state.

TABLE 6
Grades of Students in the Alpha, Beta, and Gamma Clusters

| *Metric* | *Alpha* | *Beta* | *Gamma* |
|---|---|---|---|
| Number of students (count) | 84 | 108 | 46 |
| Midterm score (%) | $M = 73.3$, $SD = 20.0$ | $M = 71.4$, $SD = 26.2$ | $M = 65.4$, $SD = 23.2$ |
| Time to complete assignment (days) | $M = 8.2$, $SD = 2.0$ | $M = 9.1$, $SD = 1.8$ | $M = 9.3$, $SD = 1.9$ |
| Karel score (%) | $M = 91.2$, $SD = 7.0$ | $M = 88.6$, $SD = 6.9$ | $M = 88.3$, $SD = 7.6$ |

furthermore, to yield deeper insights regarding fundamental misunderstandings of programming that transcend the details of one particular class.

## OVERALL DISCUSSION

We began this article by emphasizing that the fields of learning analytics and educational data mining need to extend their methodologies to detect patterns in data sets originating from *unscripted, open-ended tasks,* as they relate to skills and abilities that are gaining importance in a variety of educational settings. This requires the development of research and tools to permit (a) the use of high-frequency data, (b) multimodal capture of the entire process of generating artifacts, (c) the use of machine-learning techniques to uncover latent trajectories and patterns, and (d) predictions concerning student performance as measured by traditional metrics such as grades. As an initial step within this agenda, in this article our goal was to investigate new methodologies for examining students' behaviors as they generate open-ended computer programs.

Tackling computer science as an initial content area makes sense methodologically for reasons that we have explained before (e.g., detailed data capture is possible). It also makes sense considering the problematic track record of computer science education in the United States (McCracken et al., 2001) and elsewhere (Lister et al., 2004), and the lack of conclusive research on the factors that lead to success in introductory programming courses (Dehnadi, 2009).

One path to determining these factors has been to create taxonomies of programming styles or styles in the learning of programming (Bruce et al., 2004; Perkins et al., 1986; Turkle & Papert, 1992). This approach seemed promising because it would allow for multiple paths into programming expertise and for several different strategies for instruction, but most taxonomies have been based either on qualitative observations or on small studies ($n \sim 10$). Moreover, little research has followed up on these early efforts: Even though these categories seemed to make sense in the context of the studies, they tended to be too binary and absolute (tinkerers vs. planners), whereas in reality there are likely many more shades of gray and hybrid, dynamic forms of learning programming. But these possibly more detailed taxonomies of programming styles would be difficult to determine using traditional methods and small sample sizes. Our work is an empirical contribution toward this agenda because it expands the available analysis tools and methods and consequently increases the tractable size by orders of magnitude.

In our initial experiments, we looked at the evolution of programming behaviors across several assignments, asking whether relatively simpler quantitative measures could capture differences in these behaviors and whether these differences would be related to course performance. The main measure we used was the code update differential, which measured how much code was changed in between two snapshots. We assumed that the size of those updates (large or small) could be related to performance in the course. However, the first two experiments showed that although we were able to induce clearly distinct clusters (suggesting that there are indeed different programming styles), these clusters were very weakly related to performance. This is a somewhat surprising finding because a common assumption among computer science educators is that disciplined planning is one of the main ingredients of success in programming (two of the coauthors have been teaching these courses for several years). We found that students with very different code update patterns achieved similar grades in the course, even if there was a small (but not significant) advantage for students doing large updates. We are aware that the code update size might not be a perfect approximation of tinkering and planning, but the fact that even after successive data analysis attempts we could not find a significant correlation with course performance suggests that these two behaviors might be much less determinant for performance than computer science educators currently consider.

In Experiment 3, we hypothesized that the amount of change in students' patterns would be more determinant than update size and tinkering/planning behaviors. This approach was built on the assumption that students' update patterns would change because of good course performance, as (a) most students were novices, and thus unfamiliar with disciplined programming; and (b) one of the main learning goals was to teach programming methodology. We examined how the change in programming patterns related to formal measures such as exams and assignment grades, and we found evidence for a correlation with students' grades on assignments and, to a lesser extent, their exam grades as well. This result could indicate that a simple quantitative measure (the amount of change in code update pattern) could be significantly correlated with course performance even if we do not look at the content of the programs. This provides evidence that some of our simple measures are *detecting* latent patterns in how students learn to program. However, without more sophisticated techniques, it is very difficult for learning analytics researchers to *make sense* of these patterns, especially with data as unstructured as computer programs.

This was exactly our attempt in Study 2, in which we investigated a single assignment in considerable depth. We examined in detail students' trajectories and generated a state machine with 26 typical states for the Karel assignment. The robustness of the 26 states, as confirmed through both automated and manual analysis, suggests that *even for open-ended tasks, there are prototypical states for students to traverse, and some states are more productive than others.* The implication of this particular finding, which is novel in the literature, may be significant. We have demonstrated that *even in a completely unscripted task, and for students of diverse backgrounds and course performance, there are states and trajectories that capture the great majority of their work.* This finding could lead to the development of entirely new forms of formative and summative assessments, in which the work of the students themselves is used to generate state machines, based on realistic, ecologically valid tasks and customized for individual classrooms or student subgroups. The findings also point to the possibility of designing systems that find, in real-time, the best opportunities to offer help to struggling students in open-ended tasks, as revealed by students' trajectories.

We have also shown that there are sink states. By studying the most common transitions into such sink states, and the most successful transitions out of them, we could advise instructors on how best to help students before or after they find themselves in such states—even in real time. Ultimately, we could consider the possibility of instrumenting a software development environment to recognize such states autonomously and provide in situ guidance to students. Also, for learners, the awareness of being in a sink state could trigger productive metacognitive processes that will facilitate their return to a productive trajectory. Note that this approach is different from what is commonly called *personalized learning,* which makes predictions based on multiple-choice tests, scripted tasks, or a priori models of the learner. In our case, the model is induced from the work the students themselves generate while performing real tasks. Even though we are here just taking small steps toward that goal, it could constitute a promising research agenda for the field of learning analytics.

In Study 2, we also found a higher correlation between students' program development trajectories and their midterm scores than between their final and midterm scores. The underlying rationale is that a much larger diversity of paths exists than the final program outcomes would suggest, and thus the development path provides important information that is not available from the final product, and consequently it better predicts students' performance.

The success of the model also implies more subtle insights. The construction of the Karel state machine relied on two substantial assumptions: (a) that a dynamic time-warped save/compile event *reflected a unit of work*[9]*;* and (b) that in the generation of state transitions, the future is independent of the past given the present (the Markov assumption). The validation of the save/compile event as a unit of work is methodologically crucial, as other methods of data collection would be extremely more complex and computationally expensive (e.g., capture at regular intervals, or continuous capture). In addition, we had entertained the idea that the HMM would not be a good approach because it might miss the effect of memory, in that students might remember several of their previous states when building a subsequent step in their programs. However, the results suggest that, at least for an introductory assignment with limited complexity, the HMM assumption was correct. Building the assignment state machine based on these assumptions does not prove the assumptions are always correct, but it does demonstrate a tractable scheme for model construction that is sufficiently fine-grained to capture important trends while avoiding excess degrees of freedom in the model and overfitting the training data.

## Limitations and a Cautionary Note on Learning Analytics and Big Data in Education

We are aware that our work has several caveats. First, code was captured only for save/compile events, which was sufficient but might not have been the most accurate measure for a unit of work. Second, our current technology does not give any information about what happens when students are off task, or even how to distinguish their on-task and off-task time. An additional limitation of the work is that it required human labeling for a subset of the data for the determination of similarity between snapshots (in Study 2), which was laborious. Another technical aspect that could be problematic is that our methods were sensitive to intermediate steps in the data analysis workflow. For example, choices in how to normalize, time warp, and cluster the data could have altered the results greatly.

These limitations point to some important caveats about the endeavor of using big data in education itself. There is a popular assumption that huge data sets can reveal undisputable truths: the more data, the more discoveries. This obscures the fact that signal and noise do not scale in the same way and that there is a difference between increasing the number of observations of a given data set (the “rows”) and what parameters one is capturing in the data set (the “columns”). Current research methods in education, although limited, carry a very strong signal because the data collection is highly focused, in either qualitative or quantitative studies. Researchers spend vast amounts of time designing tests, interview protocols, and

[9]This means that our approximation of using a save/compile event as a marker for when students conclude a unit of work was correct.

focused tasks, resulting in dense data. Researchers are very explicit and intentional about the variables and parameters that the data set will contain, even if the number of observations is relatively small. As we start collecting much larger amounts of data from clickstreams, server logs, and other automated sources, the choice of what to capture is much more driven by the capturing technology (because it has to be massive), and the amount of noise grows exponentially. Thus, big data in education will only be as useful as our capacity to distinguish signal from noise, which depends on *exhaustive and complex* human intervention and on research on new ways to collect more meaningful, richer data. *Big data do not equal big signal,* so extreme care must be taken in designing data workflows to ensure a good signal-to-noise ratio, manageable dimensionality of the space, maintenance of computational tractability, and avoidance of false discovery. *This oftentimes entails more—and not less—work from researchers,* despite that fact that computers are doing the computational heavy lifting.

## CONCLUSION

We structured this article as a series of several experiments and studies of increasing complexity. This methodological rationale was to first tackle the low-hanging fruit methods for our type of data set and then move on to more sophisticated measures as needed. We believe that this approach is useful because the computational cost of different machine-learning techniques, as well as their technical complexity, can be a few orders of magnitude apart. In fact, although our first experiment took a few minutes of computation time, the number crunching in Study 2 took several days of nonstop high-performance computing power. For the learning sciences community, being able to utilize simpler tools would make learning analytics more popular and accessible. Because of this approach, and the fact that the field is in its infancy, we decided to report some results that were not significant.

In that regard, at least for our type of data set, our results show that the simpler clustering and similarity-finding techniques can reveal some types of general patterns, but they do not afford great insight into the findings. The deeper insights came from the more complex methods from Study 2. The use of open-ended, noisy data such as code snapshots has proven to be challenging, but along the way we have learned and reported a host of methodological insights and useful techniques.

The results from the two initial experiments point to the fact that the size of code updates is surprisingly not associated with course performance. However, they point to the fact that *changes* in the code update pattern are more correlated with grades. Our results are just one first step in the direction of detecting this type of change, but such a technique could be a valuable and computationally inexpensive detector of student progress, for either instructors or students.

Our second study showed that a robust state machine can be built from seemingly unrelated individual trajectories and that the states and trajectories of individual students are surprisingly universal. We found that unproductive sink states can be reliably identified and that certain types of trajectories, normally involving fewer sink states, are correlated with better student performance on exams. Study 2 also shows the generality of the patterns found in student programming paths. We analyzed the programming paths of students from two different quarters, taught by different instructors, yet we observed similar patterns in their paths in both courses. This generality indicates that such models may be more broadly applicable as a means of designing interventions that would better assist programmers, especially novices. With the necessary adaptations, this approach could be extended to other types of tasks using multimodal learning analytics (Blikstein, 2013; Worsley & Blikstein, 2013), such as the construction of robots and physical computing devices, media creation, and interaction with exploratory computer-based environments, possibly making the everyday, rigorous, large-scale use of those learning tasks much more feasible and prevalent in classrooms.

In closing, we have shown that by examining the *process* of programming rather the final product, using high-frequency data collection and machine-learning techniques, we were able to uncover counterintuitive data about students' behavior when learning to program, and we demonstrated that even in an open-ended task, patterns with better predictive power than exams can be found.

We intended this work to testify to the possibility of assessing highly personalized artifacts without diminishing the fact that they are personal. Our goal is not to use machine learning to further standardize instruction (e.g., building autograders for computer science) but to make possible more project-based work in computer science classrooms. Our two studies, in fact, show that success in computer programming is a tale of many possible pathways. Although monolithic formulas for (massive or not) teaching and learning programming are likely to fail, an approach that takes into consideration multiple starting points and multiple methods for achieving proficiency may stand a better chance. Learning to program is personal. Just as there will always be multiple solutions to a computer science problem, there should be multiple solutions to building expertise in programming, embracing diversity rather than negating it. Thus, the best approach to computer science education, echoing calls for epistemological pluralism (Turkle & Papert, 1992), is to be open to *programming pluralism.*

## ACKNOWLEDGMENT

Dedicated to Saulo Duchovni, in memoriam.

## FUNDING

This work had the support of Stanford's Lemann Center for Educational Entrepreneurship and Innovation in Brazil, Stanford's DARE (Diversifying Academia, Recruiting Excellence) Doctoral Fellowship Program, and a Google Faculty Award.

## SUPPLEMENTAL MATERIAL

Supplemental data for this article can be accessed on the publisher's website at http://dx.doi.org/10.1080/10508406.2014.954750.

## REFERENCES

Aleven, V., Roll, I., Bruce, M. M., & Kenneth, R. K. (2010). Automated, unobtrusive, action-by-action assessment of self-regulation during learning with an intelligent tutoring system. *Educational Psychologist*, *45*(4), 224–233.

Baker, R. S. J., Corbett, A. T., Roll, I., & Koedinger, K. R. (2008). Developing a generalizable detector of when students game the system. *User Modeling and User-Adapted Interaction*, *18*(3), 287–314.

Baker, R. S. J., & Yacef, K. (2009). The state of educational data mining in 2009: A review and future visions. *Journal of Educational Data Mining*, *1*(1), 3–17.

Barron, B., & Darling-Hammond, L. (2010). Prospects and challenges for inquiry-based approaches to learning. In H. Dumont, D. Istance, & F. Benavides (Eds.), *The nature of learning: Using research to inspire practice* (pp. 199–225). Paris, France: OECD Publishing.

Basawapatna, A., Koh, K. H., Repenning, A., Webb, D., & Marshall, K. (2011). Recognizing computational thinking patterns. In T. J. Cortina, E. L. Walker, L. A. S. King, & D. R. Musicant (Eds.), *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (pp. 245–250). New York, NY: ACM.

Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A., & Cook, C. (2006, April). *Tinkering and gender in end-user programmers' debugging*. Paper presented at the SIGCHI Conference on Human Factors in Computing Systems, Montréal, Quebèc, Canada.

Berland, M. (2008). *VBOT: Motivating computational and complex systems fluencies with constructionist virtual/physical robotics* (Unpublished doctoral dissertation). Northwestern University, Evanston, IL.

Berland, M., & Martin, T. (2011, April). *Clusters and patterns of novice programmers*. Presentation at the annual meeting of the American Educational Research Association, New Orleans, LA.

Blikstein, P. (2009). *An atom is known by the company it keeps: Content, representation and pedagogy within the epistemic revolution of the complexity sciences* (Unpublished doctoral dissertation). Northwestern University, Evanston, IL.

Blikstein, P. (2011a, April). *Learning analytics: Assessing constructionist learning using machine learning*. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Blikstein, P. (2011b). Using learning analytics to assess students' behavior in open-ended programming tasks. In P. Long, G. Siemens, G. Conole, & D. Gašević (Eds.), *Proceedings of the Learning Analytics Knowledge Conference* (pp. 110–116). New York, NY: ACM.

Blikstein, P. (2013, April). *Multimodal learning analytics*. Paper presented at the Third International Conference on Learning Analytics and Knowledge, Leuven, Belgium.

Booth, S. A. (1992). *Learning to program: A phenomenographic perspective* (Göteborg Studies in Educational Sciences No. 89). Göteborg, Sweden: Acta Universitatis Gothoburgensis.

Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., & Stoodley, I. (2004). Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education*, *3*, 143–160.

Burnett, M. M., Beckwith, L., Wiedenbeck, S., Fleming, S. D., Cao, J., Park, T. H., . . . Rector, K. (2011). Gender pluralism in problem-solving software. *Interacting With Computers*, *23*, 450–460. doi:10.1016/j.intcom.2011.06.004

College Board AP. (n.d.). *Computer Science A: Course description*. Retrieved from http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf

Computing curricula 2001. (2001). *Journal on Educational Resources in Computing*, *1*(3es), Article 1.

Cooper, S., Cassel, L., Cunningham, S., & Moskal, B. (2005). Outcomes-based computer science education. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (pp. 260–261). New York, NY: Association for Computing Machinery.

Cope, C. (2000). Educationally critical aspects of a deep understanding of the concept of an information system. In A. E. Ellis (Ed.), *Proceedings of the Australasian Conference on Computing Education* (pp. 48–55). New York, NY: Association for Computing Machinery.

Cutting, D. R., Karger, D. R., Pedersen, J. O., & Tukey, J. W. (1992). Scatter/gather: A cluster-based approach to browsing large document collections. In N. Belkin, P. Ingwesen, & A. M. Pejtersen (Eds.), *Proceedings of the 15th annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 318–329). New York, NY: ACM.

Dehnadi, S. (2009). *A cognitive study of learning to program in introductory programming courses* (Doctoral thesis, Middlesex University, London, UK). Retrieved from http://eprints.mdx.ac.uk/6274/

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, *39*(1), 1–38.

Dewey, J. (1902). *The school and society*. Chicago, IL: University of Chicago Press.

diSessa, A. A. (2000). *Changing minds: Computers, learning, and literacy*. Cambridge, MA: MIT Press.

Dutson, A. J., Todd, R. H., Magleby, S. P., & Sorensen, C. D. (1997). A review of literature on teaching engineering design through project-oriented capstone courses. *Journal of Engineering Education*, *86*(1), 17–28.

Dym, C. L. (1999). Learning engineering: Design, languages, and experiences. *Journal of Engineering Education*, *88*(2), 145–148.

Dym, C. L., Agogino, A. M., Eris, O., Frey, D. D., & Leifer, L. J. (2005). Engineering design thinking, teaching, and learning. *Journal of Engineering Education*, *94*(1), 103–120.

Fern, X., Komireddy, C., Grigoreanu, V., & Burnett, M. (2010). Mining problem-solving strategies from HCI data. *ACM Transactions on Computer-Human Interaction*, *17*(1), 1–22. doi:10.1145/1721831.1721834

Freire, P. (1970). *Pedagogia do oprimido* [Pedagogy of the oppressed] (17th ed.). Rio de Janeiro, Brazil: Paz e Terra.

Fuller, U., Johnson, C., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., . . . Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *SIGCSE Bulletin*, *39*(4), 152–170.

Gall, H. C., Fluri, B., & Pinzger, M. (2009). Change analysis with evolizer and changedistiller. *IEEE Software*, *26*(1), 26–33.

Ioannidou, A., Bennett, V., Repenning, A., Koh, K., & Basawapatna, A. (2011, April). *Computational thinking patterns*. Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Jadud, M. (2006). Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research* (pp. 73–84). New York, NY: Association for Computing Machinery.

Kafai, Y. B. (2006). Playing and making games for learning: Instructionist and constructionist perspectives for game studies. *Games and Culture*, *1*(1), 36–40.

Kirschner, P. A., Sweller, J., & Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, *41*(2), 75–86.

Klahr, D., & Nigam, M. (2004). The equivalence of learning paths in early science instruction. *Psychological Science*, *15*, 661–667.

Levy, F., & Murnane, R. J. (2004). *The new division of labor: How computers are creating the next job market*. Princeton, NJ: Princeton University Press.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., . . . Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin*, *36*(4), 119–150.

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008, March). Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bulletin*, *40*(1), 367–371.

Marion, B., Impagliazzo, J., St. Clair, C., Soroka, B., & Whitfield, D. (2007). Assessing computer science programs: What have we learned. *ACM SIGCSE Bulletin*, *39*(1), 131–132.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., . . . Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*, *33*(4), 1–16.

Montessori, M. (1964). *The advanced Montessori method*. Cambridge, MA: R. Bentley.

Montessori, M. (1965). *Spontaneous activity in education*. New York, NY: Schocken Books.

National Research Council. (2012). *A framework for K–12 science education: Practices, crosscutting concepts, and core ideas*. Washington, DC: The National Academies Press.

Needleman, S. B., & Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, *48*, 443–453.

Nemirovsky, R. (2011). Episodic feelings and transfer of learning. *Journal of the Learning Sciences*, *20*, 308–337.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY: Basic Books.

Papert, S. (1987). Computer criticism vs. technocentric thinking. *Educational Researcher*, *16*(1), 22–30.

Pattis, R. E. (1981). *Karel the robot: A gentle introduction to the art of programming*. New York, NY: Wiley.

Pea, R., Kurland, D. M., & Hawkins, J. (1987). Logo and the development of thinking skills. In R. Pea & K. Sheingold (Eds.), *Mirrors of mind*. Norwood, NJ: Ablex.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, *2*(1), 37–55.

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). Modeling how students learn to program. In L. A. S. King, D. R. Musicant, T. Camp, & P. T. Tymann (Eds.), *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 153–160). New York, NY: ACM.

Rabiner, L., & Juang, B. (1986). An introduction to hidden Markov models. *ASSP Magazine, IEEE*, *3*(1), 4–16.

Roll, I., Aleven, V., & Koedinger, K. (2010). The invention lab: Using a hybrid of model tracing and constraint-based modeling to offer intelligent support in inquiry environments. *Intelligent Tutoring Systems, 6094*, 115–124.

Roll, I., Aleven, V., McLaren, B., & Koedinger, K. (2011). Metacognitive practice makes perfect: Improving students' self-assessment skills with an intelligent tutoring system. *Artificial Intelligence in Education*, *6738*, 288–295.

Salton, G., Wong, A., & Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, *18*, 613–620.

Schoenfeld, A. H., Smith, J., P., & Arcavi, A. (1991). Learning: The microgenetic analysis of one student's evolving understanding of a complex subject matter domain. In R. Glaser (Ed.), *Advances in instructional psychology* (pp. 55–175). Hillsdale, NJ: Erlbaum.

Shawe-Taylor, J., & Cristianini, N. (2000). *An introduction to support vector machines and other kernel-based learning methods*. Cambridge, England: Cambridge University Press.

Siegler, R. S., & Crowley, K. (1991). The microgenetic method: A direct means for studying cognitive development. *American Psychologist*, *46*, 606–620.

Simon, B., Bouvier, D., Chen, T., Lewandowski, G., McCartney, R., & Sanders, K. (2008). Commonsense computing (episode 4): Debugging. *Computer Science Education*, *18*(2), 117–133.

Simon, B., Chen, T., Lewandowski, G., McCartney, R., & Sanders, K. (2006). Commonsense computing: What students know before we teach (episode 1: sorting). In *Proceedings of the Second International Workshop on Computing Education Research* (pp. 29–40). New York, NY: Association for Computing Machinery.

Smyth, P. (1997). Clustering sequences with hidden Markov models. In M. C. Mozer, M. I. Jordan, & T. Petsche (Eds.), *Advances in neural information processing systems 9: Proceedings of the 1996 conference* (pp. 648–654). Cambridge, MA: MIT Press.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE-10*, 595–609.

Soloway, E., & Spohrer, J. (1988). *Studying the novice programmer*. Hillsdale, NJ: Erlbaum.

Turkle, S., & Papert, S. (1992). Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, *11*(1), 3–33.

VanDeGrift, T., Bouvier, D., Chen, T., Lewandowski, G., McCartney, R., & Simon, B. (2010). Commonsense computing (episode 6): Logic is harder than pie. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (pp. 76–85). New York, NY: Association for Computing Machinery.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35.

Worsley, M., & Blikstein, P. (2013, April). *Towards the development of multimodal action based assessment*. Paper presented at the Third International Conference on Learning Analytics and Knowledge, Leuven, Belgium.