Using learning analytics to assess students' behavior in open-ended programming tasks

Conference Paper · February 2011

DOI: 10.1145/2090116.2090132 · Source: DBLP

1 author:

Paulo Blikstein
Stanford University
138 PUBLICATIONS **1,419** CITATIONS

SEE PROFILE

Using learning analytics to assess students' behavior in open-ended programming tasks

Paulo Blikstein
Transformative Learning Technologies Lab
Stanford University School of Education and (by courtesy) Computer Science.
520 Galvez Mall, CERAS 232, Stanford, CA, 94305
paulob@stanford.edu

ABSTRACT
There is great interest in assessing student learning in unscripted, open-ended environments, but students' work can evolve in ways that are too subtle or too complex to be detected by the human eye. In this paper, I describe an automated technique to assess, analyze and visualize students learning computer programming. I logged hundreds of snapshots of students' code during a programming assignment, and I employ different quantitative techniques to extract students' behaviors and categorize them in terms of programming experience. First I review the literature on educational data mining, learning analytics, computer vision applied to assessment, and emotion detection, discuss the relevance of the work, and describe one case study with a group undergraduate engineering students

Categories and Subject Descriptors
K.3.2 [**Computer and Information Science Education**]: Computer Science Education.

General Terms
Algorithms, Measurement, Performance, Language.

Keywords
Learning Analytics, Educational Data Mining, Logging, Automated Assessment, Constructionism.

1. INTRODUCTION
Researchers are unanimous to state that we need to teach the so-called "21st century skills": creativity, innovation, critical thinking, problem solving, communication, and collaboration. None of those skills are easily measured using current assessment techniques, such as multiple choice tests, open items, or portfolios. As a result, schools are paralyzed by the push to teach new skills, and the lack of reliable ways to assess them. One of the difficulties is that current assessment instruments are based on products (an exam, a project, a portfolio), and not on processes (the actual cognitive and intellectual development while performing a learning activity), due to the intrinsic difficulties in capturing detailed process data for large numbers of students.

However, new data collection, sensing, and data mining technologies are making it possible to capture and analyze massive amounts of data in all fields of human activity. These techniques include logs of email and web servers, computer activity capture, wearable cameras, wearable sensors, bio sensors (e.g., skin conductivity, heartbeat, brain waves), and eye-tracking, using techniques such as machine learning and text mining. Such techniques are enabling researchers to have an unprecedented insight into the minute-by-minute development of several activities. In this paper, we propose that such techniques could be used to evaluate some cognitive strategies and abilities, especially in learning environments where the outcome is unpredictable such as a robotics project or a computer program.

In this work, we focused on students learning to program a computer using the NetLogo language. Hundreds of snapshots for each student were captured, filtered, and analyzed. I will describe some prototypical coding trajectories and discuss how they relate to students' programming experience, as well as the implication for the teaching and learning of computer programming.

2. PREVIOUS WORK
Two examples of the current attempts to use artificial intelligence techniques to assess human learning are text analysis and emotion detection. The work of Rus et al. [13], for example, makes extensive use of text analytics within a computer-based application for learning about complex phenomena in science. Students were asked to write short paragraphs about scientific phenomena – Rus et al. then explored which machine learning algorithm would enable them to most accurately classify each student in terms of their content knowledge, based on comparisons with expert-formulated responses. However, some authors have tried to use even less intrusive technologies; for example, speech analysis further removes the student from the traditional assessment setting by allowing them to demonstrate fluency in a more natural setting. Beck and Sison [4] have demonstrated a method for using speech recognition to assess reading proficiency in a study with elementary school students that combines speech recognition with knowledge tracing (a form of probabilistic monitoring.)

The second area of work is the detection of emotional states using non-invasive techniques. Understanding student sentiment is an important element in constructing a holistic picture of student progress, and it also helps enabling computer-based systems to interact with students in emotionally supportive ways. Using the Facial Action Coding System (FACS), researchers have been able to develop a method for recognizing student affective state by simply observing and (manually) coding their facial expressions and applying machine learning to the data produced [11]. Researchers have also used conversational cues to detect student's emotional state. Similar to the FACS study, Craig et al. designed an application that could use spoken dialogue to recognize the states of boredom, frustration, flow, and confusion. They were able to resolve the validity of their findings through comparison to

emote-aloud activities (a derivative of talk-aloud where participants describe their emotions as they feel them) while students interacted with AutoTutor.

Even though researchers have been trying to use all these artificial intelligence techniques for assessing students' formal knowledge and emotional states, the field is currently benefiting from three important additions: 1) detailed, multimodal student activity data (gestures, sketches, actions) as a primary component of analysis, 2) automation of data capture and analysis, 3) multidimensional data collection and analysis. This work is now coalescing into the nascent field of Learning Analytics or Educational Data Mining [1, 3], and has been used in many contexts to measure students' learning and affect. However, in Baker and Yacef's review of its current uses, the majority of the work is focused on cognitive tutors or semi-scripted environments [2]. Open-ended tasks and unscripted learning environments have only been in the reach of qualitative, human-coded methods. However, qualitative approaches presents some crucial shortcomings: (1) there is no persistent trace of the evolution of the students' artifacts (computer code, robots, etc.), (2) crucial learning moments within a project can last only seconds, and are easy to miss with traditional data collection techniques (i.e., field notes or video analysis), and (3) such methodologies are hard to scale for large groups or extended periods of time. The cost of recording, transcribing and analysis data is a known limiting factor for qualitative researchers.

At the same time, most of previous work on EDM has been used to assess specific and limited tasks – but the "21st century skills" we need to assess now are much more complex, such as creativity, the ability to find solutions to ill-structured problems and navigate in environments with sparse information, as well as dealing with uncertainty. Unscripted learning environments are well-known for being challenging to measure and assess, but recent advances both data collection and machine learning could make it possible to understand students' trajectories in these environments.

For example, researchers have attempted to automate the collection of action data, such as gesture and emotion. Weinland et al. [15] and Yilmaz et al. [17] were able to detect basic human actions related to movement. Craig et al. [10] created a system for automatic detection of facial expressions (the FACS study). The technique that Craig et al. validated is a highly non-invasive mechanism for realizing student sentiment, and can be coupled with computer vision technology and biosensors to enable machines to automatically detect changes in emotional state or cognitive-affect.

Another area of active development is speech and text mining. Researchers have combined natural language processing and machine learning to analyze student discussions and writing, leveraging Independent Component Analysis of student conversations – a technique whose validity has been repeatedly reproduced. The derived text will is subsequently analyzed using Latent Semantic Analysis [13]. Given the right training and language model, LSA can give a clearer picture of each student's knowledge development throughout the course of the learning activity.

In the realm of exploratory learning environments, Bernardini, Amershi and Conati [6] built student models combining supervised and unsupervised classification, both with log files and eye-tracking, and showed that meaningful events could be detected with the combined data. Montalvo et al. [11], also using a combination of automated and semi-automated real-time coding, showed that they could identify meaningful meta-cognitive planning processes when students were conducting experiments in an online virtual lab environment.

However, most of these studies did not involve the creation of completely open-ended artifacts, with almost unlimited degrees of freedom. Even though the work around these environments is incipient, some attempts have been made (see 7, 8). Another of such examples is the work Berland & Martin [5], who by logging data found that novice students' developed successful program code by following one of two progressions: planner and tinkerer. Planners found success by carefully structuring programs over time, and tinkerers found success by accreting programs over time. In their study, students were generally unsuccessful if they didn't follow one of those paths.

In this paper, I will present one exploratory case study on the possibility of using learning analytics and educational data mining to inspect students' behavior and learning in project-based, unscripted, constructionist [12] learning environments, in which traditional assessment methods might not capture students' evolution. My goal is to establish a proof of existence that automatically-generated logs of students programming can be used to infer patterns in how students go about programming, and that by inspecting those patterns we could design better support materials and strategies, as well as detect critical points in the writing of software in which human assistance would be more needed. Since my data relies in just nine subjects, I don't make claims of statistical significance, but the data points present meaningful qualitative distinctions between students.

3. METHODS AND DATASET

To collect the programming logs, I employed the NetLogo [16] programming environment. NetLogo can log to an XML file all users' actions, such as key presses, button clicks, changes in variables and, most importantly, changes in the code. I developed techniques and custom tools to automatically store, filter, and analyze snapshots of the code generated by students.

The logging module uses a special configuration file, which specifies which actions are to be logged. This file was distributed to students alongside with instruction about how to enable logging, collect the log-files, and send those files back for analysis

Nine students in a sophomore-level engineering class had a 3-week programming assignment. The task was to write a computer program to model a scientific phenomenon of their choice. Students had the assistance of a 'programming' teaching assistance, following the normal class structure. The teaching assistant was available for about 3-4 hours a week for each student, and an individual, 1-hour programming tutorial session was conducted with each of the students on the first week of the study.

158 logfiles were collected. Using a combination of XQuery and regular expression processors (such as 'grep'), the files were processed, parsed, and analyze (1.5GB and 18 million lines of uncompressed text files). Below is a summary of the collected data (in this order): total number of events logged, total number of non-code events (e.g., variable changes, button presses), percent of non-code events, and actual coding snapshots.

Table 1. Number of events collected per student

Name	Events	Non-code	Non-Code %	Code
Chuck	258036	257675	99.9%	361
Che	5970	928	15.5%	5042
Leah	2836	525	18.5%	2311
Liam	4044723	4041123	99.9%	3600
Leen	253112	241827	95.5%	11285
Luca	92631	86708	93.6%	5923
Nema	3690	649	17.6%	3041
Paul	218	15	6.9%	203
Shana	4165657	4159327	99.8%	6330
Total	**8826873**	**8788777**	**99.6%**	**38096**

The overwhelming majority of events collected were non-coding events, such as variable changes, buttons pressed, and clicks. These particular kinds of event takes place when students are running or testing models – every single variable change gets recorded, what accounts for the very large number of events (almost 9 million.) Since the analysis of students' interactions with models is out of the scope of this paper, all non-coding events were filtered out from the main dataset, so we were left with 1187 events for 9 users.

For further data analysis, a combination of techniques was used. First, I developed a series of Mathematica scripts to count meaningful events within the dataset, such as number of characters, keywords used, code compilations, and types of error messages. Then, I used the resulting plots to look at particular snapshots where seemingly atypical coding activity took place – inflection points, plateaus, and sharp decreases or increases. To examine the snapshots, I developed a custom software tool, "Event Navigator" (Figure 1). The software enables researchers to go back and forth in time, "frame-by-frame," tracking students' progression and measuring statistical data.

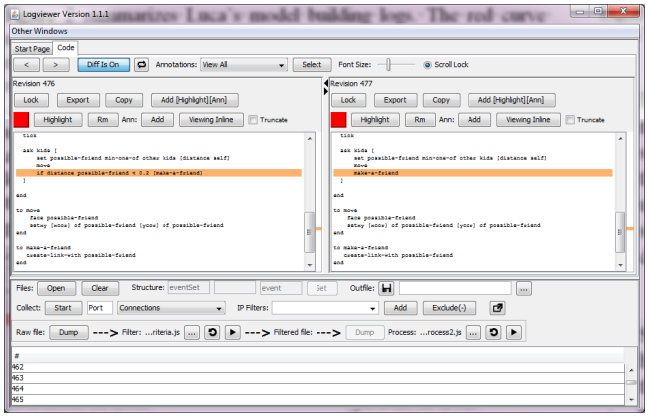


Figure 1. Screenshot of the Event Navigator software, which allows researchers to go back and forth in time, tracking how students created a computer program.

4. DATA ANALYSIS

For the analysis, I will first focus on one student and conduct an in-depth exploration of her coding strategies. Then, I will compare her work with other students, and show how differences in previous ability and experience might have determined their performance.

4.1 Coding strategies

4.1.1 Luca

Luca is a sophomore engineering student and built a scientific model in her domain area. She had modest previous experience with computers, and her grade in the class was also around the average, which makes her a good example for an in-depth analysis of her log files.

Figure 2 is a visualization of Luca's model building logs. The continuous (red) curve represents the number of characters in her code, the (blue) dots (mostly) underneath the curve represent the time between two code compilations (secondary y-axis to the right), (green) dots placed at y=1800 represent successful compilations, (orange) dots placed at y=1200 represent unsuccessful compilations (the y coordinates for those two data series are arbitrary and were chosen just for visualization purposes.) In the following paragraphs, I will analyze each of the 6 regions of the plot. The analysis was done by looking at the overall increase in character count (Figure 2), and then using the Code Navigator tool (Figure 1) to locate the exact point in time when the events happened.

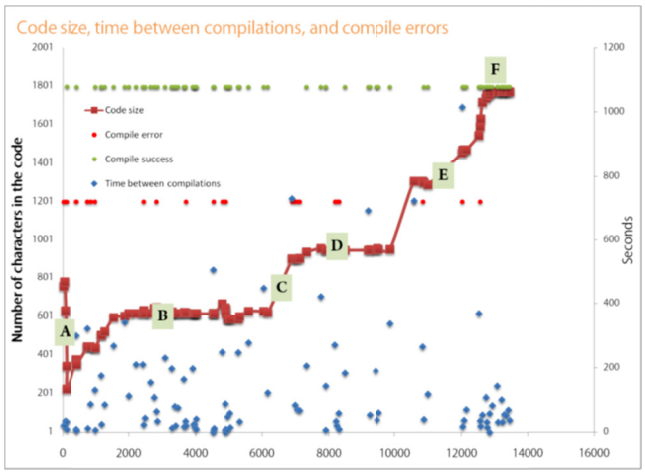
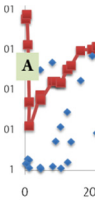


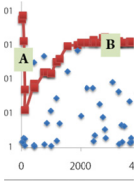
Figure 2. Code size, time between compilations, and errors, for Luca's logfiles

The following are the main coding events for Luca:

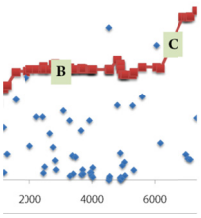
1. Luca started with one of the exemplar programs seen in the tutorial. In less than a minute, she deleted the unnecessary code and ended up with a skeleton of a new program (see the big drop in point **A**).



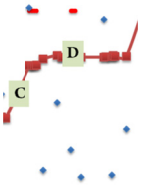
2. She spent the next half-hour building her first procedure. During this time, between **A** and **B**, she had numerous unsuccessful compilations (see the orange dots), and goes from 200 to 600 characters of code.



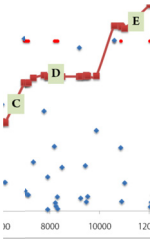
3. The size of the code remains stable for 12 minutes (point **B**), until there is a sudden jump from 600 to 900 characters (just before point **C**). This jump corresponds to Luca copying and pasting her own code: she duplicated her first procedure as a basis for a second one. During this period, also, she opens many of the sample programs within NetLogo.



4. Luca spends some time making her new duplicate procedure work. The frequency of compilation decreases (see the density of orange and green dots), the average time per compilation increases, and again we see a plateau for about one hour (point **D**).



5. After one hour in the plateau, there is another sudden increase in code size, from 900 to 1300 characters (between **D** and **E**). Actually, what Luca did was to open a sample program and copy a procedure that generated something she needed for her model. Note that code compilations are even less frequent.



6. After making the "recycled" code work, Luca got to her final number of 1200 characters of code. She then spent about 20 minutes "beautifying" the code, fixing the indentation, changing names of variables, etc. No real changes in the code took place, and there are no incorrect compilations.

Luca's narrative suggests, thus, four prototypical modeling events:

- **Stripping down** an existing program as a starting point.
- Long plateaus of no coding activity, during which **students browse other code** (or their own code) for useful pieces.
- **Sudden jumps in character count,** when students import code from other programs, or copy and paste code from within their working program.
- A **final phase** in which students fix the formatting of the code, indentation, variable names, etc.

4.1.2 Shana, Lian, Leen, and Che

Using the character count time series it is possible to examine logfiles from other students in search of similarities. In the following, I show plots from four different students (Luca, Che, Leen, and Shana, Figure 3), which include all of their activity (including opening other models—the "spikes"—note that the plot in Figure 2 did not show all of Luca's activities, but only her activities within her model, i.e., excluding opening and manipulating other models).

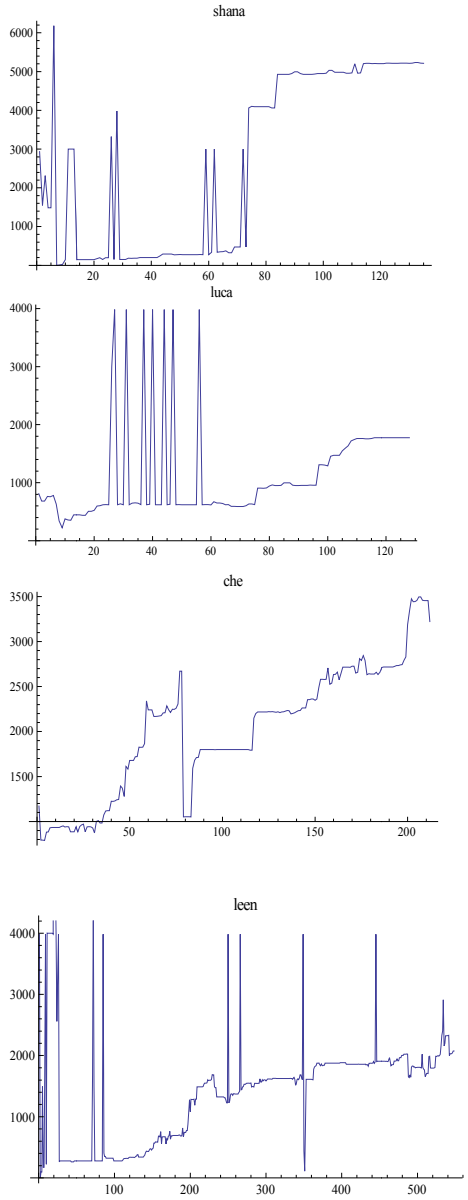


Figure 3. Code size versus time for for students: Shana, Luca, Che, and Leen. The spikes show moments in which students opened sample code.

First, let's examine Shana's logfiles. After many spikes and almost no change in the baseline character count, there is a sudden jump (at time=75) from about 200 to 4,000 characters of code. A closer, systematic examination revealed that Shana employed a different approach than Luca. After some attempts to incorporate the code of other programs into her own (the spikes), she gave up and decided to do the opposite: start from a ready-made program and add her code to it. She then chose a very well-established sample program (provided as part of the initial tutorial) and built hers on top of it. The sudden jump to 4,000 characters indicates the moment when she loaded the sample program and started to make it 'her own' by adding procedures. She seamlessly integrated the pre-existing code into her new one, adding significant new features.

Leen, on the other hand, had yet another coding style. He did open other sample programs for inspiration or cues, but did not copy

and paste code. Instead, he built his procedures in small increments by trial-and-error. In Table 2 we can observe how he coded a procedure to "sprout" a variable number of white screen elements (the action lasted 30-minutes). The changes in the code ("diffs") are indicated with the (red) greyed-out code.

Table 2. Leen's attempts to write a procedure

`to Insert-Vacancies`	`to Insert-Vacancies`
	`  sprout 2`
	`  [ set breed vacancies`
	`  set color white ] ]`
	`end`
		Initial code

`to Insert-Vacancies`
`  ask patches`
`  [ sprout 2`
`  [ set breed vacancies`
`  set color white ] ]`
`end`
	`to Insert-Vacancies`
	`  ask one-of patches`
	`  [ sprout 2`
	`  [ set breed vacancies`
	`  set color white ] ]`
	`end`
		Ask patches is introduced, and then **one-of**

`to Insert-Vacancies`
`  ask one-of patches`
`  [ sprout 1`
`  [ set breed vacancies`
`  set color white ] ]`
`end`
	`to Insert-Vacancies`
	`  ask 5 patches`
	`  [ sprout 2`
	`  [ set breed vacancies`
	`  set color white ] ]`
	`end`
		Leen experiments with **different number of patches** (1, 5, 3)

`to Insert-Vacancies`
`  ask patches-from`
`  [ sprout 2`
`  [ set breed vacancies`
`  set color white ] ]`
`end`
	`to Insert-Vacancies`
	`  loop`
	`  [`
	`  ask one-of patches`
	`  [ sprout 2`
	`  [ set breed vacancies`
	`  set color white ] ] ]`
	`end`
		Tries **patches-from** and then introduce a **loop**

`to Insert-Vacancies`
`  while`
`  [`
`  ask one-of patches`
`  [ sprout 2`
`  [ set breed vacancies`
`  set color white] ] ]`
`end`
	`to Insert-Vacancies`
	`  while n < Number-of-Vacancies`
	`  [`
	`  ask one-of patches`
	`  [ sprout 2`
	`  [ set breed vacancies`
	`  set color white ] ] ]`
	`end`
		Tries another loop approach, with a **while** command

`to Insert-Vacancies`
`  ask one-of patches`
`  [ sprout 2`
`  [ set breed vacancies`
`  set color white ] ]`
`end`
	`to Insert-Vacancies`
	`  ask 35 of patches`
	`  [ sprout 2`
	`  [ set breed vacancies`
	`  set color white ] ]`
	`end`
		Gives up looping, tries a **fixed number of patches**

`to Insert-Vacancies`
`  ask n-of Number-of-Vacancies patches`
`  [ sprout 2`
`  [ set breed vacancies`
`  set color white ] ] end`
		Gives up a fixed number, creates a **slider**, and introduces **n-of**

Leen trial-and-error method had an underlying pattern: he went from simpler to more complex structures. For example, he first attempts a fixed, "hardcoded" number of events (using the sprout command), then introduces control structures (loop, while) to generate a variable number of events, and finally introduces new interface widgets to give the user control over the number of events. Leen reported having a high familiarity with programming languages (compared to Luca and Shana), which might explain his different coding style. He seemed to be much more confident generating code from scratch instead of opening other sample programs to get inspiration or import code.

Che, with few exceptions, did not open other models during model building. Similarly to Leen, he also employs an incremental, trial-and-error approach, but we can clearly detect many more long plateaus in his graph. Therefore, based on these logfiles, seven canonical coding strategies can be inferred:

1. Stripping down an existing program as a starting point.
2. Starting from a ready-made program and adding one's own procedures.
3. Long plateaus of no coding activity, during which students browse other sample programs (or their own) for useful code.
4. Long plateaus of no coding activity, during which students think of solutions without browsing other programs.
5. Period of linear growth in the code size, during which students employ a trial-and-error strategy to get the code right.
6. Sudden jumps in character count, when students import code from other programs, or copy and paste code from within their working program.
7. A final phase in which students fix the formatting of the code, indentation, variable names, etc.

Based on those strategies, and the previous programming knowledge of students determined from questionnaires, the data suggest three coding profiles:

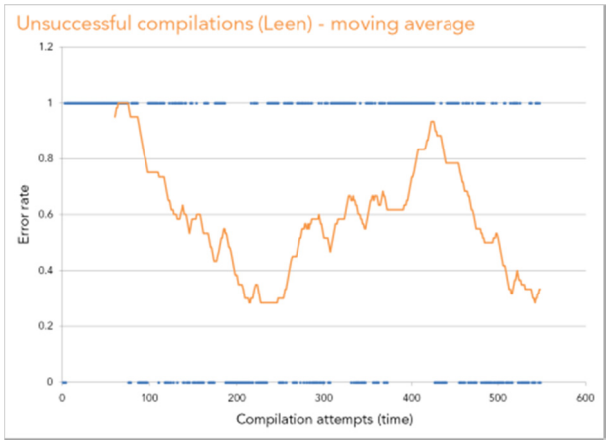
- "Copy and pasters:" more frequent use of a, b, c, f, and g.
- Mixed-mode: a combination of c, d, e, and g.
- "Self-sufficients:" more frequent use of d, e.

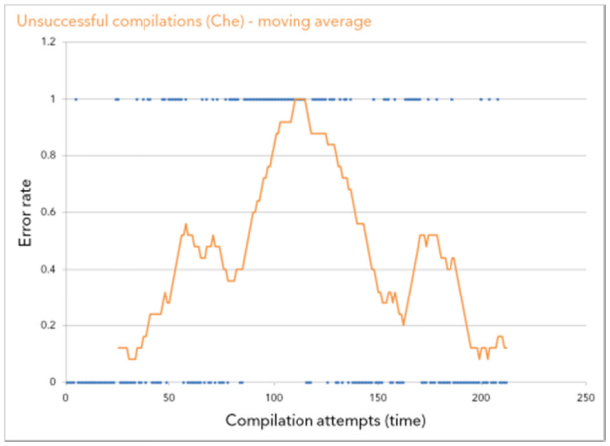
The empirical verification of these canonical coding strategies and coding profiles has important implications for design, in particular, learning environments in which engage in project-based learning. Each coding strategy and profile might demand different support strategies. For example, students with more advanced programming skills (many of which exhibited the "self-sufficient" behavior) might require detailed and easy-to-find language documentation, whereas "copy and pasters" need more working examples with transportable code. In fact, it could be that more expert programmers find it enjoyable to figure the solutions themselves, and would dislike to be helped when they are problem-solving. Novices, on the other hand, might welcome some help, since they exhibited a much more active help-seeking behavior. The data suggests that students in fact are relatively autonomous in developing apt strategies for their own expertise level, and remained consistent. Therefore, echoing previous work on epistemological pluralism, the data suggests that it would be beneficial for designers to design multiple forms of support to cater to each style (see, for example, [14]).

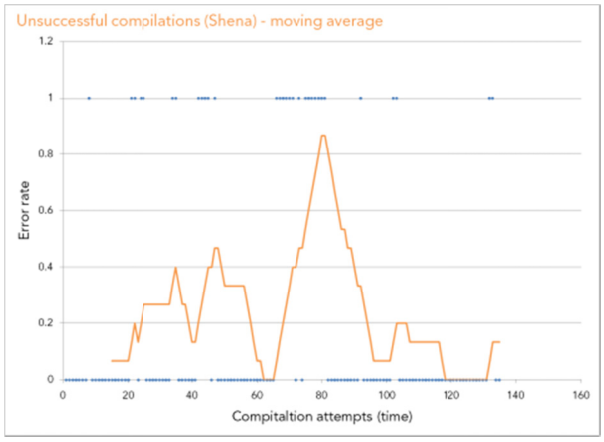
4.1.3 Code compilation

Despite these differences, one behavior seemed to be rather similar across students: the frequency of code compilation. Figure

4 shows the moving average of unsuccessful compilations (thus, the error rate) versus time, i.e., the higher the value, the higher the number of unsuccessful compilations within one moving average period (the moving average period was 10% of the overall duration of the logfile—if there were 600 compilation attempts, there period of the moving average would be 60).







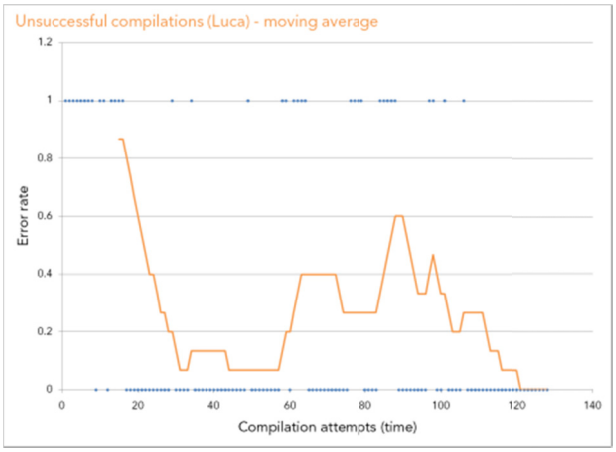


Figure 4. Error rate versus compilation attempts (time)

For all four students, after we eliminate the somewhat noisy first instants, the error rate curve follows an inverse parabolic shape. It starts very low, reaches a peak halfway through the project, and then decreases reaching values close to zero. Also, the (blue) dots on top of y=0 (correct compilations) and y=1 (incorrect compilations) indicate the actual compilation attempts. Most of them are concentrated in the first half of the activity—approximately 2/3 in the first half to 1/3 in the second half. This further confirms the data from the previous logfile analysis, in which we observed that the process of learning to program and generating code is not homogenous and simple, but complex and comprised of several different phases. In the case of code compilations, we can distinguish three distinct segments: an initial exploration characterized by few unsuccessful compilations, followed by a phase with intense code evolution and many compilation attempts, and a final phase of final touches and smaller fixes, with a lower error rate.

5. 4 CONCLUSION

This paper is an initial step towards developing metrics (compilation frequency, code size, code evolution pattern, frequency of correct/incorrect compilations, etc.) that could both serve as formative assessments tools, and pattern-finding lenses into students' free-form explorations in technology-rich learning environments.

The frequency of code compilations, together with the code size plots previously analyzed, enables us to trace a reasonable approximation of each prototypical coding profile and style. Such an analysis has important implications for the design of project-based learning environments.

First, to design and allocate support resources, moments of greater difficulty in the programming process should be identified. The data indicate that those moments happens mid-way through the project and not towards the end, as I initially suspected (given the deadline crunch anecdotally reported by many students). The proposed metrics can be calculated during the programming assignment and not only at the end, so instructors and facilitators could monitor students in real time and offer help only when the system indicates that students are in a critical zone. These zones might be detected when, for example, several incorrect compilations occur with few changes in character count, or an atypical error rate curve is identified, or when the code is changing in size too much for a long period of time.

Second, support materials and strategies need to be designed to cater to diverse coding styles and profiles. A "self-sufficient"

coder might not need too many examples, but will appreciate good command reference. Similarly, novices might benefit more from well-documented, easy to find examples with easy-to-adapt code.

By better understanding each student's coding style and behavior; we also have an additional window into students' cognition. Paired with other data sources (interviews, tests, surveys), the data could offer a rich portrait of the programming process and how it affects students' understanding of the programming language and more sophisticated skills such as problem solving.

However, the implications of this class of technique are not limited to programming. Granted, programming offers a relatively reliable way to collect 'project snapshots,' even several times per hour. But such approaches could be employed with educational software, or even with tangible interfaces, with the right computer vision toolkit.

6. LIMITATIONS AND FUTURE WORK

Due to the low number of participants, the current study does not make any claims about statistical significance. Also, because of the length of the assignment (3 weeks), some students lost part of their log files and their data could not be considered. For future studies, we will be using a centralized repository that would avoid the local storage of the log files, increasing their reliability and reduce lost data. Another limitation is that I do not log what students do outside of the programming environment, so I might mistakenly take a large thinking period with a pause.

7. REFERENCES

[1] Amershi, S., & Conati, C. (2009). Combining Unsupervised and Supervised Classification to Build User Models for Exploratory Learning Environments. *Journal of Educational Data Mining*, 1(1), 18-71.

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

[3] Baker, R. S., Corbett, A. T., Koedinger, K. R., & Wagner, A. Z. (2004). *Off-task behavior in the cognitive tutor classroom: when students "game the system"*. Paper presented at the Proceedings of the SIGCHI conference on Human factors in computing systems.

[4] Beck, J., & Sison, J. (2006). Using knowledge tracing in a noisy environment to measure student reading proficiencies. *International Journal of Artificial Intelligence in Education*, 16(2), 129-143.

[5] Berland, M. & Martin, T. (2011). Clusters and Patterns of Novice Programmers. The meeting of the American Educational Research Association. New Orleans, LA.

[6] Bernardini, A., & Conati, C. (2010). Discovering and Recognizing Student Interaction Patterns in Exploratory Learning Environments. In V. Aleven, J. Kay & J. Mostow (Eds.), Intelligent Tutoring Systems (Vol. 6094, pp. 125-134): Springer Berlin / Heidelberg.

[7] Blikstein, P. (2009). An Atom is Known by the Company it Keeps: Content, Representation and Pedagogy Within the Epistemic Revolution of the Complexity Sciences. Ph.D. PhD. dissertation, Northwestern University, Evanston, IL.

[8] Blikstein, P. (2010). Data Mining Automated Logs of Students' Interactions with a Programming Environment: A New Methodological Tool for the Assessment of Constructionist Learning. *American Educational Research Association Annual Conference (AERA 2010),* Denver, CO.

[9] Conati, C., & Merten, C. (2007). Eye-tracking for user modeling in exploratory learning environments: An empirical evaluation. *Knowledge-Based Systems,* 20(6), 557-574.

[10] Craig, S. D., D'Mello,S., Witherspoon, A. and Graesser, A. (2008). 'Emote aloud during learning with AutoTutor: Applying the Facial Action Coding System to cognitive-affective states during learning', *Cognition & Emotion,* 22: 5, 777 — 788.

[11] Montalvo, O., Baker, R., Sao Pedro, M., Nakama, A., & Gobert, J. (2010) Identifying Students' Inquiry Planning Using Machine Learning. *Educational Data Mining Conference,* Pittsburgh, PA.

[12] Papert, S. (1980). *Mindstorms : children, computers, and powerful ideas*. New York: Basic Books.

[13] Rus, V., Lintean, M. and Azevedo,R. (2009). Automatic Detection of Student Mental Models During Prior Knowledge Activation in MetaTutor. In *Proceedings of the 2nd International Conference on Educational Data Mining* (Jul. 1-3, 2009). Pages 161-170.

[14] Turkle, S., & Papert, S. (1990). *Epistemological Pluralism*. Signs, 16, 128-157.

[15] Weinland, D., Ronfard, R., and Boyer, E. (2006). Free viewpoint action recognition using motion history volumes. *Comput. Vis. Image Underst*. 104, 2 (Nov. 2006), 249-257

[16] Wilensky, U. (1999, updated 2006). *NetLogo* [Computer software]. Evanston, IL: Center for Connected Learning and Computer-Based Modeling.

[17] Yilmaz, A. and Shah, M. (2005). Recognizing Human Actions in Videos Acquired by Uncalibrated Moving Cameras. In *Proceedings of the Tenth IEEE international Conference on Computer Vision (ICCV '05) Volume 1* - Volume 01 (October 17 - 20, 2005). ICCV. IEEE Computer Society, Washington, DC, 150-157.