# METRICS FOR TIMELY ASSESSMENT OF NOVICE PROGRAMMERS*

*Jonathan P. Munson*
*Math and Computer Science Department*
*Manhattanville College*
*2900 Purchase St*
*Purchase, NY 10577*
*jonathan.munson@mville.edu*

**ABSTRACT**

We describe a study of programming behaviors of three sections of an introductory computer programming course with the goal of finding metrics for determining, early in a course, which students may be at risk. Programming activity logs generated by an instructional programming environment were used to generate a dataset of variables such as session time, number of errors, number of file edits, and other basic data. Results from correlations and linear regressions identify two simple metrics that may identify at-risk students. We also present findings that provide a basis for these models, namely, the large number of edit/compile/interpret-errors cycles that are required for beginning programmers to gain competency.

## INTRODUCTION

Instructors of introductory computer programming classes have the same mechanisms for assessing student progress as instructors of other kinds of college classes, including graded homework assignments and short quizzes. However, instructional computer programming environments that are able to collect data on student programming activity and make it available for immediate analysis offer the prospect of automated assessments that, while not being able to grade the work, may be able to provide instructors information that they would not otherwise be privy to. Instructors may be particularly interested in identifying at-risk students who are not obviously struggling. Instructors may further wish to know what aspects of programming at-risk students are

struggling with. For example, they may find it valuable to know if a student is having trouble making effective use of error messages, or if the student is having difficulty keeping track of nested elements.

To know what automated assessments to program, we must know what kind of activity predicts success, or failure, in introductory computer programming courses. In this paper we report on a study involving three sections of an introductory computer programming course. We identify two activity metrics that predict students' final grades for the course, and could thus potentially alert instructors to the need for interventions with at-risk students.

## RELATED WORK

There have been many studies of novice programming behavior using data gathered from programming environments or programming-assignment submission environments. Ihantola et al provide a comprehensive survey in [8].

### Detecting Students in Difficulty

Murphy et al [10] developed a tool, Retina, that provides instructors a view of students' performance on assignments, including the number of compilations, the number of errors, and the most common error a student is encountering. The information is intended to enable instructors to better help students requesting help, but is also used, through rules, to automatically provide recommendations to students, such as explanations of errors that are frequently occurring.

Carter and Dewan [2][3] developed tools for automatically determining when programmers are in difficulty based on command sequences logged from instrumented programming environments. The actions are at a level of interactions with the environment, such as run-program, debug-program, set-breakpoint, invoke-content-assist, insert-text, and others.

Instead of looking at activity patterns, Alammary, Carbone, and Sheard [1] developed a tool that looks at students' activity primarily on the code itself (identifying syntax errors and missing expected features) and secondarily their interaction with the environment (idle time).

### Quantifying Error-Handling to Predict Performance

Jadud [6][7] developed the technique of quantifying how programmers manage an error from one compilation to the next. Research by Rodrigo et al [11] and Tabanao, Rodrigo & Jadud [13] has further developed this technique, looking more closely at the handling of specific errors, and doing within-group comparisons of at-risk, average, and high-performing students. Both studies found that total errors encountered were negatively correlated with performance. The two studies differed on whether time between compilations was correlated.

Watson, Li, and Godwin, for the portion of their study [14] involving data logged from a programming environment, also used the compilation-pairs technique. Their

results indicate that how long an error remains unresolved over successive compile attempts, and the amount of time spent working on errors, are both significant predictors of student performance. This research also employed questionnaire-based assessments and found that, overall, the programming-based metrics performed better as predictors.

In an earlier work of our own [9], we took a more behavior-oriented approach, and focused on how effectively novice programmers interpreted error messages. We were able to correlate this behavior with performance on programming assignments.

Based on the large number of studies with the same objective-identifying automated metrics that reliably predict student performance-it is evident that the idea is a popular one. The first phase of this work, identifying the metrics, seems to have had good results. The work described here, in part, serves to bolster these results, offers additional description of how students actually work in introductory programming courses, and provides a specific quantification of one aspect of student performance.

## METHODOLOGY

We collected data on the programming activity of 58 students in three sections of an introductory computer-programming course over the entire course (in two separate semesters). The programming language used was Java. Grade levels ranged from freshman to senior. The students' majors of study ranged over all of the majors offered by the college; few were computer science majors. Data was anonymized before being saved to persistent storage.

Students used an instructional programming environment we developed for use in our introductory programming courses. The system, which we call "Codework," is designed to streamline the assignment workflow, offer a simplified development environment, and provide data to support empirical studies of interventions. Codework offers students a browser-based user interface in which they develop their code and receive compilation and execution results. File storage, compilation, and execution are performed on a server. Codework offers instructors their own Web-based interface to review and execute student code.

Program-editing events are captured continuously from the JavaScript-based editor and sent to the server at intervals, piggybacked on automatic file-save requests. At the server, they are stored to a database. Compilation events, triggered explicitly when the student clicks the "Compile" button, are likewise logged to the database. For each event we capture the program source, the list of compiler error messages, the file name, the assignment the student is working on, a timestamp, the course section the student is in, and an ID for the student that is separate from and independent of the college-assigned ID.

After courses are completed and grades are assigned we pull the compilation data from the database, create a data set for statistical analysis, and derive several additional variables. A total of 40,316 compilation events were logged by the system; after removing data from students who did not complete the course and removing data for compilation events from in-class activities (which often involved transcribing work modeled by the instructor), 21,288 data points remained.

This data was aggregated to produce per-assignment statistics and per-student statistics, shown in Table 1.

Table 1. Variables computed for each student's compilation events.

| | |
|---|---|
| sumDiffs | The total number of file-differences computed for successive versions of files the students has developed. |
| numCompiles | The total number of compilation attempts made by the student over the course. |
| diffsPerCompile | The average number of differences per compilation. |
| numErrs | The total number of compilation errors over the course. |
| numErrsPerCompile | The average number of errors per compilation. |
| totalSessionTime | The total amount of time spent using the programming environment by the student (in minutes). [1] |
| medianSessionLen | The median length of all the student's programming sessions (in minutes). |
| meanSessionLen | The mean length of all the student's programming sessions (in minutes). |

Each error reported by the compiler was identified as one of 51 error types (matching the error message against fixed message patterns) and classified as either syntactic or semantic. A separate dataset was created with a variable for each error type, the value being how many errors of that type were reported for that compilation. In addition to these 51 variables, each record contains the subject ID, section ID, timestamp, assignment ID, filename, the compilation number within the assignment file and the compilation number within the entire course.

## RESULTS

We first looked at students' performance on some basic metrics: total number of compilations, total number of edits, total number of errors, number of errors per compilation, number of differences per compilation, total time spent in the programming environment, and mean and median session lengths. For each metric, we group the students by their final grade: A, B, C, or D/F. Table 2 shows the distribution of the grades of the students in this study.

[1] Less than actual time; does not include time spent before first compile in a session, and does not include time spent after last compile and before logout.

Table 2. Grade Distribution

| # A's | # B's | # C's | # D/F's | Max | Min | Median | Mean |
|---|---|---|---|---|---|---|---|
| 18 | 22 | 8 | 10 | 100.0 | 41.1 | 85.6 | 82.6 |

**Total number of compilations**

Figure 1 shows a box plot of the total number of compilations per student, over the entire course.

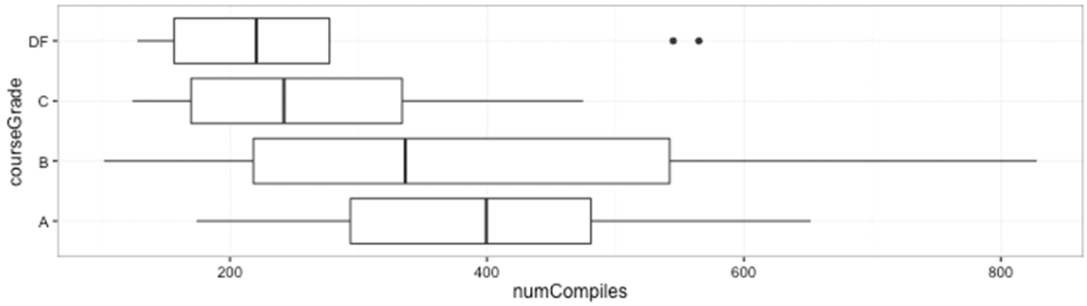


Figure 1. Distribution of total number of compilations per student.

We performed a linear regression modeling grade as a function of number of compilations and found that it does significantly predict GPA ($p < 0.01$). The coefficient was 0.027, indicating that, in this simple model, for every 37 (1/0.027) compiles, the student's overall course grade goes up 1 point.

The figure, combined with the linear regression results, shows that number of compiles is a better predictor of poor performance than of high performance. That is, it could potentially be used to identify students at risk. We will discuss this further in the DISCUSSION section.

**Total number of errors and total number of edits**

Figure 2 and Figure 3 shows the distributions of the total number of edits and total number of errors. As noted above, the number of edits in any one compile is approximated by computing file differences between the contents of the file after the last compilation and the contents of the file at the current compilation.

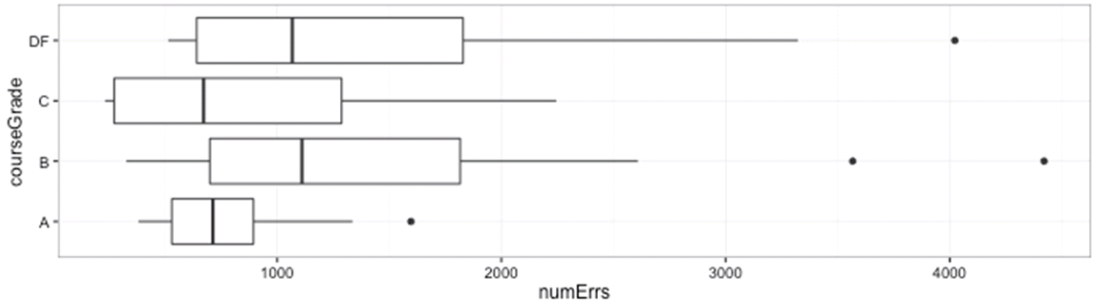


Figure 2. Distribution of total number of errors.

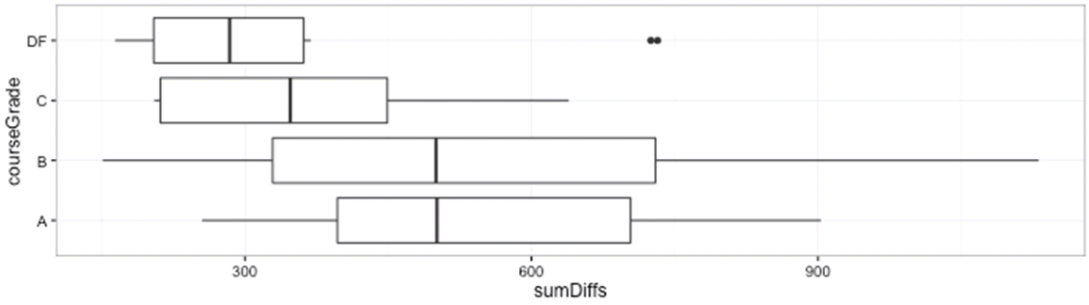


Figure 3. Distribution of total number of edits.

Figure 2 displays an apparent anomaly, which is that the median number of errors of 'A' students is about the same as for 'C' students, and the median for 'B' students is about the same as for 'D'/'F' students. So by itself, number of errors does not significantly predict performance, as shown in Model 0, Table 3 (the coefficient is not significant). However, it is a significant factor when the number of edits is added to the model, as shown in Model 1, Table 3. The number-of-errors coefficient becomes significant, and the fraction of variance explained goes from .03 to .34. According to Model 1, for a given number of edits, the fewer the errors, the higher the grade. Likewise, for a given number of errors, the more the edits, the higher the grade.

Model 2 includes total session time in addition to number of errors and number of edits. Adding total session time significantly reduces the magnitude of the association between number of edits and course grade (.041 is reduced to .021, nearly halved). This coefficient reduction is significant, ($z=2.03$, $p < .05$) [4], and explains almost 40% of the variance in the course grade.

Table 3. Regressions of course grade on number of edits, number of errors, and total session time.

| | Model 0 | | Model 1 | | Model 2 | |
|---|---|---|---|---|---|---|
| | Coeff. | Std. Err | Coeff. | Std. Err | Coeff. | Std. Err |
| Intercept | 85.5*** | 2.99 | 73.2*** | 3.44 | 70.8*** | 3.54 |

| | | | | | | |
|---|---|---|---|---|---|---|
| numberErrors | -0.002 | 0.002 | -0.01*** | 0.002 | -0.01*** | 0.002 |
| numberEdits | – – | – – | 0.04*** | 0.008 | .02* | 0.012 |
| totalSessionTime | – – | – – | – – | – – | .020** | 0.010 |
| | | | | | | |
| R-squared | 0.03 | | 0.34 | | 0.39 | |
| *p<.05, **p<.01, ***p<.001, ****p<.0001 | | | | | | |

**Total session time**

Figure 4 shows the distribution of total session time (in minutes). The positive correlation shown in Figure 9 is also evident in this figure. Nonetheless, when we included it in a linear model for course score, it made the numberEdits factor less significant. Given the high correlation between total session time and number of edits, it is likely that total session time explains the number of edits.

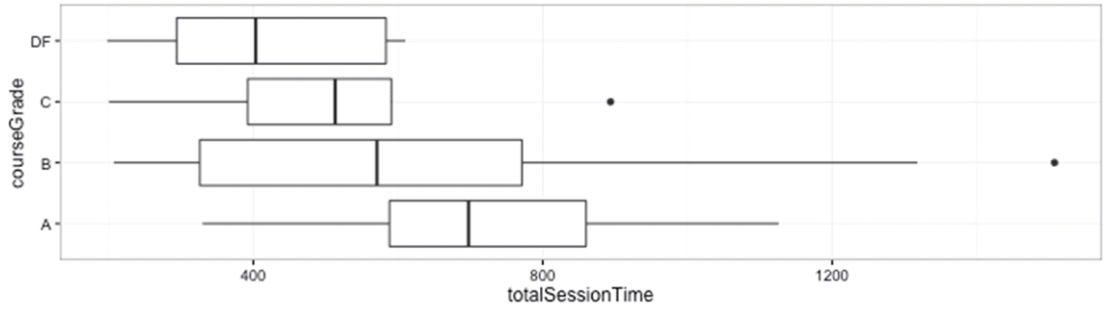


Figure 4. Distribution of total session time (in minutes).

**Number of errors per compilation**

Figure 5 reveals that the number of errors per compilation, unlike the total number of compilations, is a somewhat better predictor of good performance than it is of poor performance. However, as the correlation matrix in Figure 9 shows, it is strongly negatively correlated with course score.

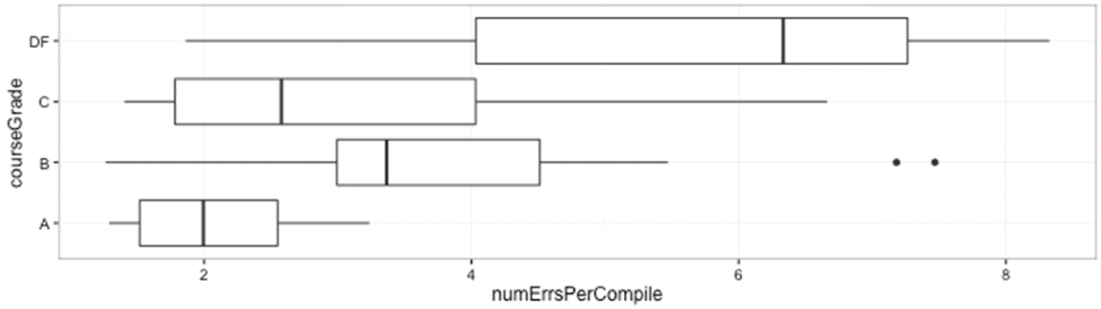


Figure 5. Number of errors per compilation.

**Total number of file edits**

Figure 6 shows that students in the CDF category made significantly fewer edits than students in the B and A categories. This suggests it may be used as an indicator of at-risk students.

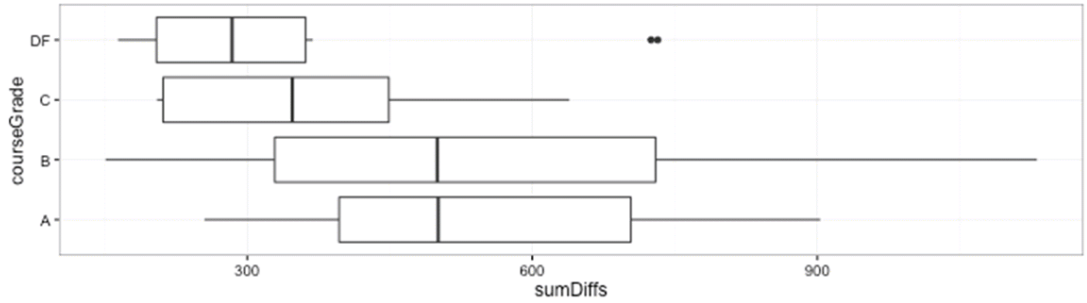


Figure 6. Total number of file edits ("sumDiffs").

**Number of file edits per compilation**

Number of file edits per compilation, as shown in Figure 7, has minimal ability to predict course score.

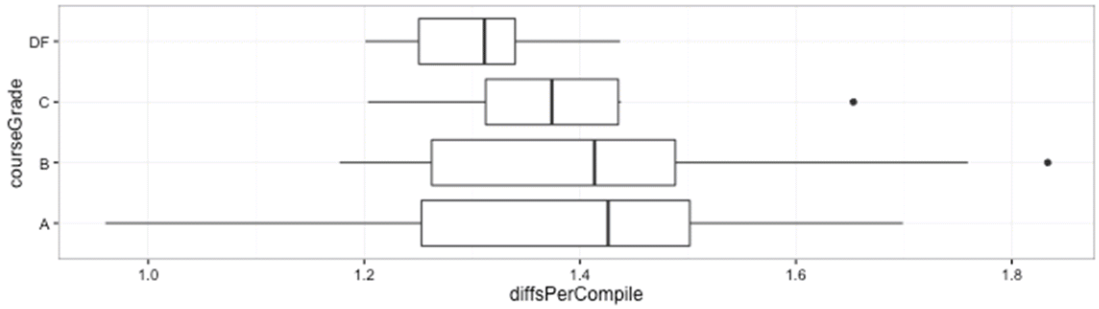


Figure 7. Number of file edits per compilation.

**Median session length**

Figure 8 shows that 'A' students have a higher median session length than the other classes, but Figure 9 shows that the overall correlation is not high. Therefore, this variable is, by itself, not a good predictor of performance.

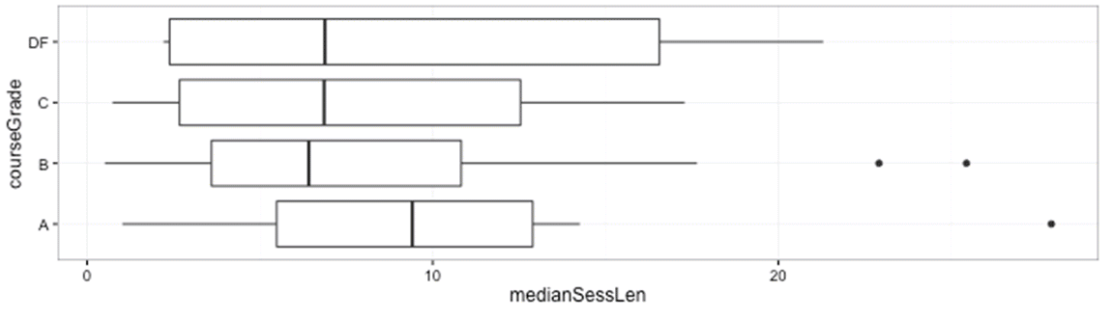


Figure 8. Distribution of median session length (in minutes).

## Correlations

Figure 9 shows the correlations between each pair of variables in Table 1.

| | courseScore | sumDiffs | numCompiles | diffsPerCompile | numErrs | numErrsPerCompile | totalSessionTime | medianSessLen | meanSessLen |
|---|---|---|---|---|---|---|---|---|---|
| courseScore | 1.00 | 0.36 | 0.34 | 0.15 | -0.16 | -0.60 | 0.42 | 0.02 | 0.17 |
| sumDiffs | 0.36 | 1.00 | 0.98 | 0.06 | 0.59 | -0.11 | 0.84 | 0.18 | 0.45 |
| numCompiles | 0.34 | 0.98 | 1.00 | -0.13 | 0.62 | -0.08 | 0.88 | 0.21 | 0.47 |
| diffsPerCompile | 0.15 | 0.06 | -0.13 | 1.00 | -0.16 | -0.07 | -0.21 | -0.11 | -0.08 |
| numErrs | -0.16 | 0.59 | 0.62 | -0.16 | 1.00 | 0.64 | 0.50 | 0.02 | 0.12 |
| numErrsPerCompile | -0.60 | -0.11 | -0.08 | -0.07 | 0.64 | 1.00 | -0.15 | -0.17 | -0.28 |
| totalSessionTime | 0.42 | 0.84 | 0.88 | -0.21 | 0.50 | -0.15 | 1.00 | 0.24 | 0.53 |
| medianSessLen | 0.02 | 0.18 | 0.21 | -0.11 | 0.02 | -0.17 | 0.24 | 1.00 | 0.80 |
| meanSessLen | 0.17 | 0.45 | 0.47 | -0.08 | 0.12 | -0.28 | 0.53 | 0.80 | 1.00 |

Figure 9. Correlation matrix for per-student metrics

## Error Count as Function of Compilation Number

As compilation records are processed we assign a per-student sequence number to each compilation record. So the number 1 represents each student's first compilation attempt, and 200 represents each student's 200th compilation attempt. We are interested in this number as a common "clock" that we can use to measure progress.

The figures below show the number of two types of errors for compile numbers 1 through 1000, for three classes of students. On the left are students who received an 'A' for the course and on the right are students who received a 'C'. The Y axis shows the total number of errors for a compile number divided by the number of students who reached that compile number. On each plot a Loess curve through the points is overlaid. The two types of errors are two of the most common compilation errors in our dataset: "… expected" (SYN0) and "illegal start of expression" (SYN3).

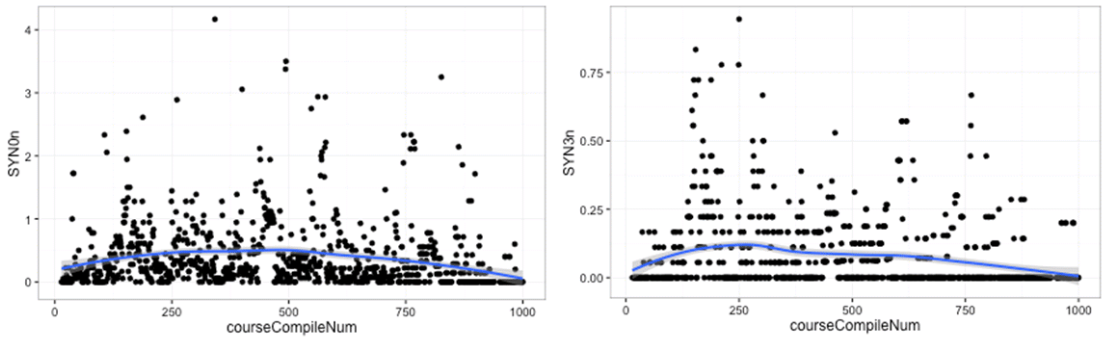


Figure 10. Error Count vs. Compile Number for SYN0 and SYN3 Errors, 'A' Students

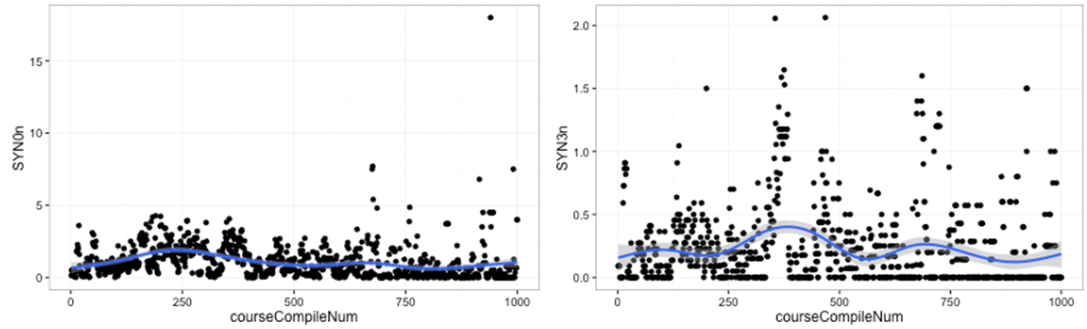


Figure 11. Error Count vs. Compile Number for SYN0 and SYN3 Errors, 'B' Students

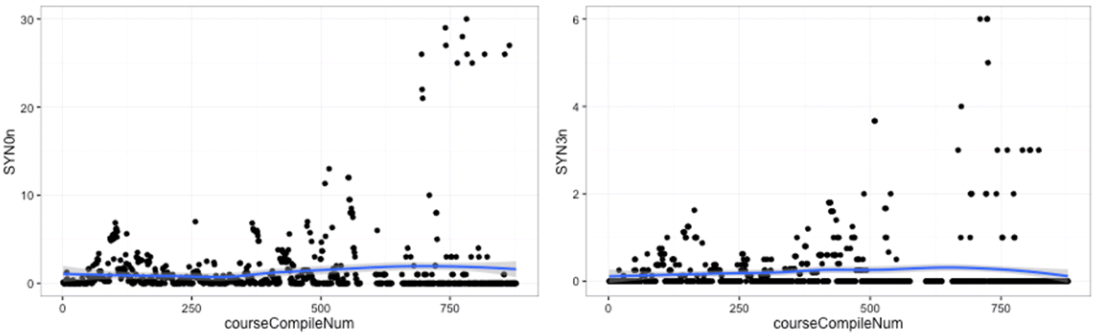


Figure 12. Error Count vs. Compile Number for SYN0 abd SYN3 Errors, 'C' Students

Note that because the error count at each compile number is divided by the number of students who reached that compile number, the error-count values at the higher compile numbers are not averaged as much as those at the lower compile numbers, and thus there tends to be more spread in the plots at the higher compile numbers, particularly in Figure 12. The Loess curve is included to estimate the trend.

Comparing the SYN0 error counts for the three classes of student, we see that for 'A' students, the average error count (using the Loess curve) is around 0.5 per compile up

to the 500th compile, and then begins to decline. 'B' students are at about 1 error per compile at the 500th compile, and do not decline much after that. The average for 'C' students does not very much from 2 errors per compile. Similar results hold, although at different levels, for SYN3 errors. The average error count for 'A' students shows a decline beginning at around the 500th compile.

## DISCUSSION

The objective of this study was to determine which metrics would be useful to guide instructors in their decisions about which students may need extra attention. If we wish to focus on students at risk, then our results indicate that we should employ two metrics: (1) the number of errors per compile, and (2) a metric based on Model 1 discussed above (Table 3), where course score is a linear combination of total number of edits and total number of errors. There are other models with some predictive ability but these two were the most robust.

The first metric is somewhat easy to understand as a measure of a student's ability to overcome errors. The second metric, course score as a function of total number of edits and total number of errors, seems to be a kind of measure of effective work. It measures not simply the amount of work nor the number of errors, but rather whether the work is effective (at removing errors).

The error-count results suggest a basis for these models-students, even the best, find it challenging to produce syntactically correct programs. Not until midway through the course do the error counts for the best students begin to decline. My analyses suggest that they succeed because (1) they work harder (a greater number of compile/edit/interpret-errors cycles, (2) they are more effective at removing errors, at least in part because (3) they learn more quickly how to interpret error messages [9].

Our initial objective in undertaking these studies was to determine metrics that could be used to detect students whose work patterns matched those who had done poorly in the course before. However, we found such wide variability in the predicting variables-see Figure 1, Figure 2, and Figure 3-that, given the data we have, we feel that any classifier based on these metrics would have an unsatisfactorily low precision. Therefore, we feel that a more fruitful approach would be to inform students of how their work metrics compare to those of students before them, possibly using the same box-and-whisker graphics as shown here. This would be somewhat similar to the Retina system's [10] ability to inform students about their performance (average number of compiles, average number of errors, etc.) relative to other students on particular assignments. However, our displays would compare their metrics to those of 'A' students, 'B' students, etc., based on the data from past semesters. We believe this would be highly meaningful to grade-conscious students.

For this instructor, simply being able to tell students that they should expect that it may take upwards of 500 compiles before they begin to be "fluent" with syntax is a useful result. It may help to reduce discouragement among some students. Finally, this analysis puts hard numbers to the concept of "effort."

**ACKNOWLEDGMENTS**

Funding from the Manhattanville College Math and Computer Science Department paid for the cloud services that Codework runs on. This support is gratefully acknowledged.

**REFERENCES**

[1] Alammary, A., Carbone, A., & Sheard, J. Implementation of a smart lab for teachers of novice programmers. In *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123* (2012, January). pp. 121-130.

[2] Carter, J., Dewan, P., Pichiliani, M. Towards Incremental Separation of Surmountable and Insurmountable Programming Difficulties. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15).* (2015), 241-246.

[3] Carter, J., Dewan, P. Design, implementation, and evaluation of an approach for determining when programmers are having difficulty. In *Proceedings of the 16th ACM international conference on Supporting group work.* (2010, November). pp. 215-224.

[4] Clogg, C. C., Petkova, E., Cheng, T. (1995). Reply to Allison: More on comparing regression coefficients. *American Journal of Sociology*, 100: 1261-1305.

[5] Helminen, J., Ihantola, P. and Karavirta, V. 2013. Recording and Analyzing In-Browser Programming Sessions. *Koli Calling '13*. (2013), 13-22.

[6] Jadud, M.C. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education.* 15, 1 (2005), 25-40.

[7] Jadud, M. C. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research* (2006, September). pp. 73-84.

[8] Ihantola, Petri, et al. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. (2015) URL: http://www.cs.tut.fi/~ihantola/iticse2015dataminingwg.pdf

[9] Munson, J., Schilling, E., Analyzing Novice Programmers' Response to Error Messages, In *Journal of Computing Sciences in Colleges*, Vol. 31, No. 3, January 2016.

[10] Murphy, C., Kaiser, G., Loveland, K. and Hasan, S. 2009. Retina: helping students and instructors based on observed programming activities. *ACM SIGCSE Bulletin*. (2009), 178-182.

[11] Rodrigo, M. M. T., Baker, R. S., Jadud, M. C., Amarra, A. C. M., Dy, T., Espejo-Lahoz, M. B. V., Lim, S.A.L., Pascua, S.A.M.S.,, Sugay, J.O. & Tabanao, E. S. Affective and behavioral predictors of novice programmer achievement. In *ACM SIGCSE Bulletin* Vol. 41, No. 3 (2009, July), pp. 156-160.

[12] Tabanao, E. S., Rodrigo, M. M. T., & Jadud, M. C. Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research* (2011, August). pp. 85-92.

[13] Watson, C., Li, F. W., & Godwin, J. L. (2014, March). No tests required: comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 469-474).