Syntax Error Based Quantification of the Learning Progress of the Novice Programmer

Alireza Ahadi
University of Technology
Sydney, Australia
Alireza.Ahadi@uts.edu.au

Raymond Lister
University of Technology
Sydney, Australia
Raymond.Lister@uts.edu.au

Luke Mathieson
University of Sydney
Australia
Luke.Mathieson@uts.edu.au

ABSTRACT

Recent data-driven research has produced metrics for quantifying a novice programmer's error profile, such as Jadud's error quotient. However, these metrics tend to be context dependent and contain free parameters. This paper reviews the caveats of such metrics and proposes a more general approach to developing a metric. The online implementation of the proposed metric is publicly available at http://online-analysis-demo.herokuapp.com/.

CCS CONCEPTS

• **Social and professional topics → Computer science education**;

KEYWORDS

programming, replication, syntactic errors, semantic errors, student mistakes

1 INTRODUCTION

For decades, students who were learning to program submitted their assignments on paper. Thus, the only artifact available for analysis was a student's final program. In recent years however, with the advent of web-based systems for teaching programming, it is now possible to study the student behavior that culminates in the final program.

This paper reviews a number of parameters which should be considered in designing a metric of analyzing students error profiles and introduces a metric for quantifying compilation error based learn-ability of programming novices.

2 BACKGROUND

Dynamically accumulating data such as programming process data has only recently gained researchers' attention [1, 3, 4, 8, 11, 14]. In 2006, Jadud suggested quantifying a student's tendency to create and fix errors in subsequent source code snapshots (i.e. compiled states). He called it the Error Quotient (EQ) and defined it as a function of error messages: for all compilation events, each pair of consecutive compile events are ranked based on the result of the compilations. The EQ score ranges between zero and one, with zero indicating strong ability in correcting syntactic mistakes.

Rodrigo et al. [12] proposed an alternative version of Jadud's EQ by considering the location of the error and the edit made to the code in order to fix the error. They found that, in their context, the correlation between the EQ and the midterm score of an introductory programming course was strong and statistically significant ($r=-0.54$; $p<0.001$). Watson et al. [14] proposed an improvement to Jadud's EQ called *Watwin*, and found that with this improvement the correlation increased from $r=0.44$ to $r=0.51$. They also noted that a simple measure, the average amount of time that a student spends on a programming error, is strongly correlated with programming course scores ($r=-0.53$; $p<0.01$).

The EQ algorithm and its derivatives have been investigated in different contexts. Carter et al. [5] compared their proposed algorithm, the Normalized Programming State Model (NPSM), to both EQ and the Watwin scores, however the investigated programming languages were different: C/C++ rather than Java. That study suggested that context may have great impact on the result of the EQ algorithm. The importance of the effect of context on such data-driven models has also been demonstrated [2], where both EQ and Watwin measures showed poor performance when the data was limited to the first week of a programming course. In the study performed by Peterson et al. [10] it was shown that EQ and its derivatives are sensitive to context. Becker [4] introduced a new metric to quantify repeated errors called the repeated error density (RED). He compared that to Jadud's EQ and showed that RED had advantages over EQ including context independency and practicality for short sessions.

3 METRIC DESIGN

The following subsections explore different considerations in the metric design process and how we evaluated the performance of the metric.

3.1 Metric Characteristics

3.1.1 General Attributes. The strength of a metric in quantifying learning aptitude is dependent on a variety of parameters. We argue

that the operationalization of the metric can be best achieved by comparing an individual's performance on one programming task with another programming task (See Section 3.2). Also, a strong metric should demonstrate *replicability*. That is, a strong metric should report a similar error profile/progress for the same individual on two nearly identical programming tasks done in quick succession.

Other attributes to be considered when designing a metric are ease of implementation, applicability, context independence, no use of free parameters, minimal sensitivity to the bias, and population independence.

The last item is concerned with the fact that the amount of progress that a student makes should not be compared to a cohort of students but with the previous learning state of the same individual. We believe that comparing an individual to a whole student cohort is related to the applicability of a particular teaching strategy and the difficulty level of a particular exercise, rather than the student's learning ability.

3.1.2 Language Independence. From a computer science perspective, the major difference between the two programming courses is often the language taught. Many universities use Java as the primary programming language for their CS1 course, while some universities use Python, C or even Perl. Peterson et al. [10] investigated the application of the EQ based algorithms in multiple contexts. They reported the impact of the language taught on the outcome of the EQ metric and its derivations. In general, programming languages can be divided into two groups: compiled languages and interpreted languages. For compiled languages, programming errors are either compile-time errors or run-time errors. In an interpreted language, both of these error types are revealed only at the point of interpretation. In general, compiled languages allow more robust error detection for certain classes of errors (particularly with modern integrated development environments), whereas interpreted languages typically require the code to actually be *run* to detect these errors. This fundamentally is a matter of differences in the data generated in different environments, however a strong metric for quantifying learning to code should not be significantly affected by such changes.

The definition of what is called an error is also in many cases arbitrary: many syntax errors in one language are not regarded as errors in other languages, but rather presented as warnings. Even if one has fixed all compilation errors, the compiler of some languages (such as C and C++) may still give you "warnings". These warnings won't keep the code from compiling (unless the compiler is asked to treat warnings as errors). These warnings could be treated as errors in another programming language.

A metric for profiling students' learning ability to code should not be trained based on the features of a specific language, but rather be based on what all programming languages have in common, which is something that from now on we refer to as the *universal features*.

3.1.3 Distribution Independence. In the metric proposed by Watson et al. [14] the time required to fix the error encountered by each novice is compared with the rest of the population. Based on the median and the standard deviation of the time spent on fixing the given error, a penalty value of 1, 15 or 25 is included in the calculation of the final Watwin score. The fundamental issues with comparing one's quantified attribute of coding (in this case, the time spent to fix the code) to the rest of the population are that **a)** the given parameter plays a big role in the algorithm, **b)** regardless of other contributing features to the element of time, the comparison obscures the hidden effect of those features, and **c)** the comparison also obscures other features which are not present in the algorithm which can distinguish one particular student from others. For example, consider a student who has spent a lot more time on the code compared to others, but has not necessarily failed in completing the exercise successfully and in fact has learned how to write correct code. Including time and comparing it with the whole population is in fact very dependent on the context and could be very misleading. For example, the majority of students in the authors' institute finish the exercises very quickly and leave the test room. This is not due to the fact that they are good programmers, but mainly because their primary strategy of handling the code is rote learning. Our analysis shows that these students in fact get lower marks in the final exam compared to those who in fact show a longer period of engagement with the code. In a way, the interpretation of the amount of time spent on the code contradicts its application in the Watwin algorithm: those who finish their programming task most quickly tend to achieve lower scores in the final exam and those who spend more time (who would be penalised in the Watwin algorithm) exhibit deeper learning, which would normally be considered a more desirable trait.

3.1.4 Cross-Context Variations in the Parameters Used in the Construction of the Metric. Some features used in the construction of a metric are highly affected by the changes in the context. For example, the element of time is directly affected by the condition under which the data is generated. Under exam conditions, a novice wouldn't necessarily have enough time to sit and think about how to fix the code in a particular exercise. However, if the data is collected from students working on their assignments in an unsupervised condition, then it is almost impossible to say what exactly the novice has been doing. One could have left the machine after facing an error, gone to have some lunch, while another could have been talking about it on the phone to a friend. Thus, such attributes are not good contributors in the construction of a strong metric as they are greatly impacted by contextual (and some time incidental) settings.

3.1.5 Ad-Hoc Parameters. The EQ algorithm considers two penalties in the calculation of the EQ score. A penalty score of 8 is considered if two consecutive compilation events both present syntax errors. Another additional 3 penalty points are added if both errors are of the same type. Page 233 of the Jadud's thesis [7] reads:

> We began with one set of parameters for this algorithm, and used a semi-exhaustive search to find a better set of parameters in the surrounding space. Those are now the default parameter set employed by the calc-score function.

This indicates that the values of 3 and 8 are intentionally selected to best suit the context in order to increase the performance of the

algorithm. We argue that such parameters should not play a role in the construction of the algorithm. There is also a danger of overfitting which can greatly affect the performance of such algorithms.

3.1.6 Validity of the Operationalization. Operationalization is the process through which an abstract concept is translated into measurable variables. Operationalized concepts are related to theoretical concepts but are not coincident. The major problem with operationalization is the problem of validity. How can one be sure that the operational measurement still measures the theoretical concept? There are no certain ways in which validity can be "tested" because of the break between theory and practice (empirical data) that is integral to the quantitative research tradition. Further, to what extent can a single measure quantify the concept? For example, what is the difference between a novice with an EQ score of 0.4 and a novice with a score of 0.6? A robust validated metric should consider representing a measurement which is truly reflective of how much the novice has learned to code.

3.1.7 Number of Compiles. We aimed to investigate the characteristics of the number of compiles as a quantifying feature of learning ability. Number of compiles has a relatively high correlation with the number of line edits in our data (Pearson correlation coefficient = 0.68). We examined the number of compiles made by a cohort of students (N = 281) while doing a particular programming task.

The minimum number of lines required to complete the given program skeleton was 35. We noticed that a total number of 80 (Total N = 281) students did not compile their code more than twice, and 41 of those students compiled their code only once.

For all these novices, the compilation was done at the end of a long series of edit events. We hypothesized that two types of students would complete their exercise successfully with only a limited number (≤ 2) of compiles. The first category is those students who are good programmers. The second is those who have memorized the solution. These students tend to finish the programming task much faster as they avoid critical thinking while completing the given code. For those 81 students who took only one or two compiles to complete the task, we looked at their score on questions from the end-of-semester exam. The exam questions we looked at were worth a total of 8 marks, and required students to write code for tasks they had not seen before.

Students with a total number of compiles of no more than two have either scored 8 of 8, or scored very low (either 0 or 1). More interestingly, among those students, those who had finished their programming task much faster were the students who scored zero or one. These results support our intuition that the 81 students with a very few number of compiles are either good programmers, or they have memorized the solution. We concluded that the number of compiles is not a good parameter to be included in the body of the algorithm as it does not distinguish poor performing students from strong students.

3.1.8 Number of Edits. We analyzed the correlation of the number of line edits made during the programming task with the compile to edit ratio. Interestingly, among those who have a higher number of edits to complete an exercise, the compile/edit ratio is either very low, or very high. When examining the source code of the two categories, we realized that those with a high number of edits and a low number of compiles are struggling to fix the semantic mistakes in the code, and those with a high number of edits and a high number of compiles are in fact trying to fix the syntax of the code. This led us to the conclusion that the high number of edits could sometimes be an indicator of dealing with a semantic error, especially when the number of error compiles is low. Automated identification of the semantic mistakes requires a solid definition of what is defined as semantically *correct*. However, regardless of the type of mistake, the error to edit ratio seems to be an indicator of whether a student is challenged by a particular programming task or not.

Given the reasons in this section, we argue that a strong metric should not be dependent on the absolute frequency/count of its parameters, but to some extent must reflect the relative ratio between these parameters on an individual basis.

3.1.9 Dependency on the Data Point Features. In most cases, there are quite a few inevitable variations in the data which have roots in the sample size, the context, and other unknown contributing factors which can affect the distribution of data points and consequently the interpretations derived from the data. This also affects the mathematical model. For instance, the Rasch model [6], one of the most widely used methods for performing one parameter item response theory, is limited to a set of data-related assumptions. Equal item discrimination, uni-dimensionality, and low susceptibility to guessing are some of the Rasch analysis *requirements*. In a Rasch analysis, such requirements are usually handled by eliminating items that appear to violate the assumption. We believe that a strong model for capturing the ability to code should be free of such data related assumptions.

3.2 Conceptual Framework

Several research programs emphasize the role of failure in learning and problem solving. Impasse-driven learning [13] and productive failure [9] are examples of such theories which benefit from the learner's mistakes (impasses to be be more precise) to improve learning outcomes. From a computer science perspective, automated measurements of the *progress* of the individuals within the above-mentioned theories is very close to the concept underpinning intelligent tutoring systems. In such systems, automated domain model construction is deployed with the core aim of constructing a student model that tracks a student's progress through the programming task with the goal of constructing a final domain model. Yudelson [15], for example, investigated automatic generation of user models for the assignment-grading system deployed in a set of introductory programming classes. (Note though that the commonality between the above mentioned theories and the work done by Yudelson is in the tracking of progress among different states, rather than the sharing of the same theoretical perspective.)

Each semantic/syntactic programming mistake is a small impasse on the way to successfully completing a programming task. Thus, in the spirit of impasse-driven learning [13] and productive failure [9], the basic concept in our approach is the measurement of a student's progress from one compilation event to the next.

3.3 Design

For the reasons reviewed, we aimed to design a metric which *a)* is not dependent on the frequency of its contributing parameters, but the relative changes of these parameters from one task to another, *b)* is not sensitive to number of edits, number of compiles or time, *c)* compares a suitable feature extracted from two similar exercises for each novice, rather than comparing a novice with the population as a whole, and *d)* is not sensitive to context, coding strategy or problem solving style. The metric is based on the relative amount of errors generated in a second programming task compared to a first task, where the the two tasks are similar. This metric measures how much one student has learned from the syntactic mistakes committed in an task by comparing student's error ratio to the second task, where the two programming tasks are very similar in nature. By "similarity" we mean the topics covered in the two tasks should be on same data structures and spanning the same topics so the difference between the causes of different types of errors generated in the two exercises is minimized.

The proposed metric is calculated in three steps. In the first step, the syntax error ratio of the novice in task A (E_a) is calculated. In the second step, the syntax error ratio of the novice in task B (E_b) is calculated, where task B is similar to task A. Finally, the logarithmic fold change of the two values is calculated based on the following equation: %bigskip

$$\mathrm{L} = \log_2(E_b) - \log_2(E_a) = \log_2 \left(\frac{E_b}{E_a} \right)$$

The syntax error ratio is defined as the frequency of number of syntactic errors, divided by the total number of compilation events. If L is a negative number, it can be concluded that the novice has learned from task A, when *learning* in this context is defined as making less mistakes while coding. The metric can also be more specific, targeting particular types of errors. In this mode, the primary goal is to quantify the novice's ability in handling a specific type of error:

$$\mathrm{L}(k) = \log_2(E_b(k)) - \log_2(E_a(k)) = \log_2 \left(\frac{E_b(k)}{E_a(k)} \right)$$

Where k refers to a specific error type, $E(k)$ refers to the error to compile ratio of k, and L(k) gives how much the novice has learned from encountering k in task A, by comparatively measuring it in task B. It is possible to group a set of errors which are similar in nature, are due to specific reason, or can be related to one specific line/chunk of code to calculate the metric. Grouping different types of errors together however is not recommended as different error types have different significance.

For comparative analysis purposes, the z score can be used to identify those who are performing much poorer/better compared to the rest of the population. The z-score of a novice with a L value can be calculated based on the following equation:

$$z_s = \frac{L_s - \bar{L}}{\sigma_L}.$$

Those novices who have a z score value of less than -1.65 or more than $+1.65$ are those who have either shown significant ($p < 0.05$) (relative to the population) improvement or decline respectively.

3.4 Metric Evaluation

To evaluate our methodology, we first calculated the L value of a total number of 209 participants on two exercises (exercises *A* and *B*) which satisfy the criteria briefed in section 3.3. The median of E_a and E_b were 0.51 and 0.37 respectively, with an L value of -0.44 which indicates fewer syntactic mistakes in the second exercise. We repeated the same task on a different group of subjects ($N = 274$) in another semester and we observed the same result ($E_a = 0.48$, $E_b = 0.34$, and $L = -0.49$). This means that the number of errors made in the second programming task is fewer than in the first.

The L values of around 67% of students were less than zero. We could infer from this that most students demonstrate learning between the exercises. However do not know for sure if an individual with a negative value for the metric has indeed learned something and those with a positive L value have not. To investigate this, we decided to investigate the association between performance on a set of selected final exam questions and the L value. We limited the analysis to those with a z-score of greater than $+1.65$ and less than -1.65. This makes the focus of the analysis the identification of poor/strong students with a high level of certainty.

Table 1 reviews the percentage of the students labeled as either strong or poor by the L metric who answered different questions of the final exam correctly. As can be seen in the majority of cases, the percentage of students who did better in the final exam questions is higher with negative L values (p value of the Barnard's test < 0.03). Barnard's test is computationally intensive, and is not as widely used as Fisher's Exact test. However, it is known that it is a more powerful test especially for the 2 by 2 contingency table scenarios and small sample sizes.

Note that the primary approach for tackling the programming tasks for a considerable number of students the authors' institute is memorization. As discussed, these students tend to compile their code very few times hence the very nature of the data generated does not offer enough compilation events. This leads to a worst case scenario for testing the performance of the proposed metric. The up side of this scenario however is that it is testing the performance of the proposed metric under a more restricted condition.

Table 1: The percentage of students in each category identifted by L metric who answered different ftnal exam questions correctly.

Question	$z < -1.65$	$z > 1.65$
Q40a	.76	.75
Q40b	.73	.62
Q40c	.64	.37
Q42a	.52	.75
Q42b	.7	.62
Q42c	.82	.87
Q42d	.39	.37
Q42e	.54	.37

4 DISCUSSION

To better model student's intentionality behind each line edit and learning progress through analysis of the source code snapshot

data, extensive knowledge of the *data* is required. What is the collected data in fact representing? For example, the majority of syntax errors made by novices are due to a missing semicolon. Is forgetting to add a semicolon (the second most common syntactic mistake according to our data) in fact a fundamental problem? Does failing to add a semicolon simply manifest carelessness, rather than a lack of understanding? Is remembering to add a semicolon alone in fact an indicator of learning to code? Better measurement of learning to code requires consideration of intentionality.

There are some limitations to the proposed model. First, the operationality of this metric is dependent upon the range of activities that the student has been doing. From the moment that the first programming task is completed to the moment that the second programming task is started, the student's activities are unknown. For example, has s/he been practicing, or has s/he been getting help from other students or resources? Hence, an optimal act of *operationalization* would be achieved when the model is trained based on two consecutive programming tasks which are due in the same programming session, with the minimum time delay between them. Also, the proposed method would work best if the programming task is focused on a small programming task with a focus on a specific topic, rather than covering a range of different topics. here, "small" means a chunk of code like a method which targets a specific topic such as conditional statements. On the other side, like other metrics of the same category, this metric is more sensitive to the strategy the student uses, but not necessarily sensitive to how much they have truly learned. That is, it aims to measure how they develop a strategy to not make a specific type of mistake, but it does not necessarily reflect how much avoiding the mistake has helped them learn better. Finally, the extreme similarity of the two programming tasks is a requirement of the proposed method as some errors encountered in one of the exercises might not be committed in the second exercise due to non-similarity of the exercises.

5 CONCLUSION

Compilation errors have been shown to be an indicator of students struggling with programming. We have introduced a new metric to measure progression in coding using source code snapshot metadata. This metric is not context dependent, language dependent and does not suffer from ad hoc parameters. The proposed metric can be calculated in an error type specific manner hence focusing on quantifying the progress in tackling a particular type of error. An online implementation of the metric is available at http://online-analysis-demo.herokuapp.com/.

REFERENCES

[1] Alireza Ahadi, Arto Hellas, and Raymond Lister. 2017. A contingency table derived method for analyzing course data. *ACM Transactions on Computing Education (TOCE)* 17, 3 (2017), 13.

[2] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. 2015. Exploring Machine Learning Methods to Automatically Identify Students in Need of Assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*. ACM, New York, NY, USA, 121–130. https://doi.org/10.1145/2787622.2787717

[3] Alireza Ahadi, Raymond Lister, and Arto Vihavainen. 2016. On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 218–223.

[4] Brett A Becker. 2016. A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 296–301.

[5] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. 2015. The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*. ACM, New York, NY, USA, 141–150. https://doi.org/10.1145/2787622.2787710

[6] Ronald K Hambleton and Win J van der Linden. 1997. *Handbook of modern item response theory*.

[7] Matthew Jadud. 2006. *An Exploration of Novice Compilation Behaviour in BlueJ*. Ph.D. Dissertation.

[8] Matthew C Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*. ACM, 73–84.

[9] Manu Kapur. 2014. Productive failure in learning math. *Cognitive Science* 38, 5 (2014), 1008–1022.

[10] Andrew Petersen, Jaime Spacco, and Arto Vihavainen. 2015. An exploration of error quotient in multiple contexts. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. ACM, 77–86.

[11] Leo Porter, Daniel Zingaro, and Raymond Lister. 2014. Predicting student success using fine grain clicker data. In *Proceedings of the tenth annual conference on International computing education research*. ACM, 51–58.

[12] Maria Mercedes T Rodrigo, Emily Tabanao, Ma Beatriz E Lahoz, and Matthew C Jadud. 2009. Analyzing online protocols to characterize novice Java programmers. *Philippine Journal of Science* 138, 2 (2009), 177–190.

[13] Kurt VanLehn, Stephanie Siler, Charles Murray, Takashi Yamauchi, and William B Baggett. 2003. Why do only some events cause learning during human tutoring? *Cognition and Instruction* 21, 3 (2003), 209–249.

[14] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2013. Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*. IEEE, 319–323.

[15] Michael Yudelson, Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Investigating Automated Student Modeling in a Java MOOC. In *Proceedings of The Seventh International Conference on Educational Data Mining 2014*.