Measuring Students' Performance on Programming Tasks

Tomáš Effenberger
Masaryk University
Brno, Czech Republic
tomas.effenberger@mail.muni.cz

Radek Pelánek
Masaryk University
Brno, Czech Republic
pelanek@fi.muni.cz

ABSTRACT

Large scale learning systems for introductory programming need to be able to automatically assess the quality of students' performance on programming tasks. This assessment is done using a performance measure, which provides feedback to students and teachers, and an input to the domain, student and tutor models. The choice of a good performance measure is nontrivial, since the performance of students can be measured in many ways, and the design of measure can interact with the adaptive features of a learning system or imperfections in the used domain model. We discuss the important design decisions and illustrate the process of an iterative design and evaluation of a performance measure in a case study.

INTRODUCTION

A key part of adaptive learning systems is student modeling, which provides the foundation for the intelligent behavior of these systems. Research in student modeling focuses mainly on the intricacies of modeling temporal dynamics of learning and complex relations among knowledge components [3]. Relatively little attention has been given to the basic input to student modeling: the summary of a students' performance on a given task. In most cases, modeling approaches consider only binary information about correctness. Several authors consider richer information, e.g., a partial credit based on students' use of hints [8], high speed high stakes rule based on response times [2], or classification of incorrect answers based on the frequency of errors [5]. However, such approaches are not utilized in the current mainstream student modeling approaches [3].

More complex measures of students' performance are useful particularly in domains where students' interaction with a task is complex and relatively long. We focus specifically on introductory programming. In this case, a student can spend several minutes solving a programming exercise. Using just the information about the correctness of the attempt would lead to the loss of potentially useful information about the student's knowledge state. Performance measures in such case can range from simple heuristics based on summary statistics (e.g., response time, number of code edits), through more complex algorithms that consider all individual interactions with the task, to a recurrent neural network over a series of program snapshots embeddings [7].

Utilizing detailed information about students' interaction with a task (as done in [7]), is potentially powerful, but complicates the development of adaptive learning systems. Even in the case of introductory programming, different activities (turtle graphics, a robot on a grid, text processing) have significantly different interaction modes and each of them would require complex research into suitable modeling of students' performance.

On a general level, we propose to use discrete performance measures with just a few distinct values. We argue that this should be enough to capture the main differences in students' performance, and at the same time, it makes the development of adaptive learning systems much more viable as it makes student modeling independent of the idiosyncrasies of a particular educational task. We show that even for a simple introductory programming exercise the design of a performance measure is nontrivial and involves interaction with domain modeling. Our results are directly relevant for many other introductory programming activities that are today used on very large scale; e.g., *Hour of Code* activities are solved by millions of student every year [9].

MEASURES OF STUDENTS' PERFORMANCE

A performance measure is a function that takes as an input a log data on a student's interaction with a particular task and outputs a summary measure of the performance. A performance measure can be seen as a model of a student's performance on a task. The key difference with student modeling (as commonly used) is that student modeling uses data on a sequence of tasks and takes into account temporal dynamics (learning) along the sequence.

Overview of Possible Approaches

Previous research on student modeling avoids the need for a nontrivial performance measure by either using a binary success as a trivial performance measure [3], or by using raw observational data directly to train a model [7]. However, both of these extreme approaches have severe disadvantages.

Binary success contains too little information. Solving a programming task takes much more time than, e.g., answering a multiple choice question about a geography fact. Furthermore, most tasks in a well-designed system for teaching introductory programming are eventually solved—failing to solve a task after several minutes of trying is frustrating and decreases the

L@S '19 June 24–25, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6804-9/19/06.
DOI: https://doi.org/10.1145/3330430.3333639

motivation of students, so it is important to avoid it. For example, the two problems from Hour of Code with published data have success rates 81.0% and 97.8 % [7]. Therefore, binary success delivers infrequent and not very informative signal about the student.

Specifically, the binary success is clearly insufficient to guide recommendation decisions: if the student spent 10 minutes solving the first task in the system, he should be recommended a different task than a student who solved this task in 10 seconds. The binary success is also insufficient to optimize tutor models: if the student solves the task, it could still be too easy, or too difficult for the student, so we do not know whether the recommendation was appropriate.

The other extreme is to use all observational data as the input to, e.g., a student model [7]. However, the lack of unified input makes such complex models challenging to reuse, even in a single system with multiple exercises with different observational data.

Between these two extremes, there is a broad spectrum of options for performance representations, ranging from discrete performance levels (most interpretable, least information), through a single-value continuous performance, to a multidimensional performance embedding (least interpretable, most information). Beyond the binary success, it becomes unclear how to interpret the values and how to compute them. Also, most of the currently used student models are designed to work with binary performance as their input, so these must be adapted in order to work with the more complex performance representations.

Discrete Performance Measures

For introductory programming tasks, a few discrete performance levels provide a good tradeoff between the interpretability of the binary success and the precision of the continuous performance. The precision can be increased by adding more performance levels (better approximating the continuous performance) at the cost of either weaker interpretability, or more demanding annotation process.

As the first step, we propose to extend the currently prevalent binary success to four performance levels: *failed* $<$ *poor* $<$ *good* $<$ *excellent*. Already in this case, the interpretation is ambiguous, and it is even less clear for more levels or a continuous representation. One possible interpretation of these levels is based on the student's experience: *poor* performance means that the task was too difficult for the student, *good* performance indicates an appropriate task, and *excellent* performance too easy task. This interpretation makes the measured performance useful as a signal for tutor model optimization.

DESIGN OF A PERFORMANCE MEASURE

So far, we have argued that discrete performance measures can be useful. However, it is not clear how to design a suitable performance measure for a given type of exercise. Does it matter which performance measure we choose, or do all reasonable choices lead to similar measurements?

To explore this question, we performed a case study of designing a performance measure for *RoboMission*, an adaptive learning system for introductory programming. *RoboMission* is a programming game—a variation on the commonly used theme "robot in a grid", which is often used for introductory programming (e.g., in the Hour of Code activities). Students create programs using a block-based programming interface and can execute their program at any time to see its effect in the game environment. The game covers sequences of commands, loops, and conditional statements; see [1] for details. There are 85 tasks divided into linearly ordered hierarchical problem sets (9 levels, each with 3 sublevels). Each sublevel contains about 3 similar tasks, which practice the same concepts. The data used for the analysis consist of 62,500 *task sessions* (a series of uninterrupted interactions with a task) attempted by 3,800 students in total, and about 1 million of program snapshots taken after each edit and execution. Most task sessions (81%) are successful.

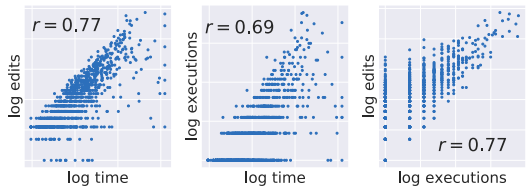

Figure 1. Relationship between number of code edits, number of executions and solving time.

Choice of Input Data

In the programming tasks, the quality of the solution is usually an important criterion. However, in introductory programming, the diversity of the correct solutions is lower, because the programs are short and use only a few commands known to the students. The structure of the solution is often further constrained; for example, in RoboMission, most tasks impose a limit on the length of the program. Consequently, most solutions are similar and thus do not carry useful information for performance evaluation.

Therefore, we only consider time, number of edits and number of executions as criteria for the measurement. Since all these measures are highly skewed, we use them log-transformed. We explore correlations between these features to see if we can omit any of them. The global correlations (i.e., computed over all task sessions) are high—over 0.69 for all pairs of measures (Figure 1). However, they are not high within all individual tasks; about 20 % of the tasks have correlations between the time and a click-based signal below 0.5. In contrast, the correlation between edits and executions is high for all tasks, indicating that it is sufficient to use only one of these click-based signals.

In this case study, we explore performance measures based on a single criterion. The thresholds for good and excellent performance can be set either globally, per problem set, or even per each task. Another decision is whether to set these thresholds manually, learn them from labeled task sessions, or find them using collected observational data. We illustrate the diversity by comparing three distinctly different measures: (1) *Execution Count Measure*, that assesses as excellent the task sessions with only one execution, and good performance with

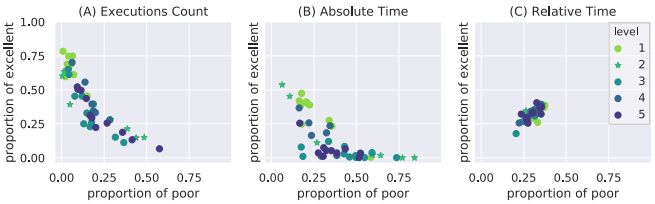

Figure 2. Scatterplots of task difficulties for three performance measures. Each task and is placed according to the proportion of *poor* and *excellent* performances. The rest of the performances is *good*.

at most 5 executions, (2) *Absolute Time Measure* with thresholds per each problem set (obtained by a procedure described in section 3.2), and (3) *Relative Time Measure* with thresholds per each task computed as multiples of median solving time on the task. These measures behave very differently (Figure 2), so the choice of the measure is an important decision.

Iterative Improvement of Thresholds

How to set the thresholds for the *Absolute Time Measure*? Standard supervised techniques are not directly applicable, because reasonable thresholds must be chosen even before any performance data are collected, and later improvements should require only a few task sessions to be labeled manually. Furthermore, the collected data are not identically and independently distributed due to a mixture of biases such as a personalized recommendation of the next task, learning, self-selection bias, and attrition bias [4].

The accuracy of a performance measure can also be negatively affected by flaws in the domain model[1]. Figure 2B illustrates an interaction between the performance measure and the domain model. There are tasks in the level 2 (marked as stars in the plot) at both ends of the difficulty spectrum. It could be an issue with the specific performance measure, but in this case, it is actually an issue with the domain model: the level 2 contains both easy and extremely tricky tasks, that should not be given to the students so early.

When deciding that it is the domain model that should be fixed, not the performance measure, we used an implicit assumption about a reasonable progression of the thresholds for a smooth learning experience. Specifically, the thresholds should not change arbitrarily wildly, but rather they should slowly and gradually increase. We made this assumption explicit by adding a constraint on the threshold progression, requiring a constant increase of the thresholds (in the log-time scale) between the corresponding sublevels of consecutive levels, and similarly a constant increase between consecutive sublevels of any given level. Such a constraint leads to a gradually growing curve, with periodic drops after the end of each level, an oscillation behavior recommended for the challenge intensity in games [6].

This constraint also reduces the number of parameters that need to be set initially by the task authors. In order to make them easier to estimate for people, we made each parameter

[1]In this case study, the term *domain model* corresponds to the mapping between tasks and problem sets. Most of the discussion is, however, relevant also for more general meaning of the term.

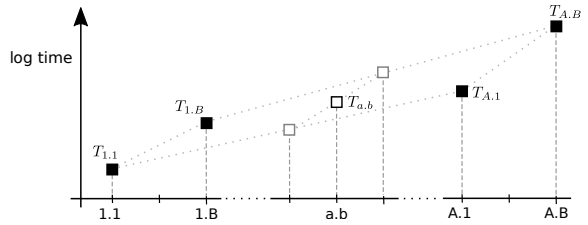

Figure 3. Thresholds parametrization for domains with linearly ordered hierarchical problem sets. The filled black squares correspond to four parameters that need to be set for each performance level (e.g., for good performance), empty squares denote linearly interpolated thresholds.

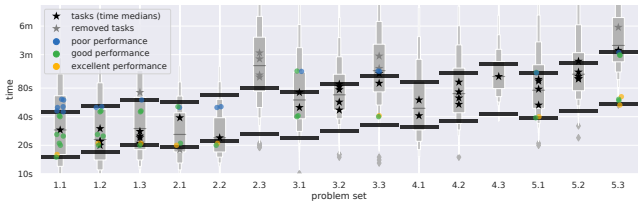

Figure 4. Threshold progression plot. For each problem set, it shows the time thresholds for good and excellent performance (thick black lines), distribution of solving times (gray plots), median solving time for each task (stars), and labeled task sessions (colored dots). The distribution of solving times is visualized using extended box plots (letter-value plots).

to correspond to a threshold in a specific sublevel. This leads to 4 parameters for good performance: thresholds for the first and last sublevel of the first and last level (and analogically 4 parameters for excellent performance). All other thresholds are interpolated linearly in a log-transformed 2D space with levels and sublevels as the coordinates (Figure 3). Specifically, let $T_{a.b}$ be a threshold for level a, sublevel b; A the total number of levels, B the number of sublevels in each level. Based on the manually specified thresholds $T_{1.1}, T_{1.B}, T_{A.1}, T_{A.B}$, we specify threshold $T_{a.b}$ as $(1-\alpha)(1-\beta)T_{1.1} + (1-\alpha)\beta T_{1.B} + \alpha(1-\beta)T_{A.1} + \alpha\beta T_{A.B}$, where α and β represent progress towards the final level, and towards the final sublevel of the current level, respectively: $\alpha = (a-1)/(A-1)$, $\beta = (b-1)/(B-1)$. The specific parametrization depends on the structure of the domain (in our case, linearly ordered hierarchical problem sets); different domains would need a different parametrization. The general guiding principle is to choose a parametrization with only a few parameters, all of them with a straightforward interpretation.

Once some performance data are collected, we can visualize them in *threshold progression plot* (Figure 4), which reveals discrepancies between the thresholds and the observed performances. To find out whether the discrepancies are caused by wrong thresholds or wrong domain model, a few task sessions randomly selected around a specified threshold can be labeled. The task author then decides whether changing thresholds or rather changing domain model is a more suitable remedy. For example, even tasks practicing the very same concepts should not be in the same sublevel (i.e., sharing the same thresholds) if one task needs only 2 blocks, and the other 20, resulting in longer solving times. In our case, there are a few outlying tasks, which violate the homogeneity of sublevels and should be removed or modified, and there is even a whole

outlying sublevel 2.3. At the end of this iterative procedure of joint optimization of both performance measure and domain model, the sublevels should be roughly homogeneous, and the time thresholds should reasonably separate the excellent performances from the good, and the good from the poor.

Evaluation of Performance Measures

To evaluate which of the performance measures is the best, we use a combination of multiple qualitative and quantitative methods. Two diagnostic visualizations that we have found particularly useful are the already discussed *task difficulties scatterplot* (Figure 2) and *threshold progression plot* (Figure 4).

Execution Count Measure is the most benevolent, with some tasks having over 75 % of excellent and nearly no poor performances. The *Absolute Time Measure* is much more strict. The labeled task sessions in the *threshold progression plot* suggest its strictness is justified, and the distribution of performances indicates that the difficulty of the tasks is increasing too fast. The *Relative Time Measure* is from the definition rather agnostic to the task difficulty, and hence it fails to detect the too difficult problematic tasks; tasks in the introductory levels have a similar proportion of the excellent performances as the tasks in the more advanced levels.

We also measured agreement with human annotations on a randomly selected sample of 70 task sessions. Two annotators (authors of the paper) first labeled the task sessions independently, and then they resolved discrepancies between them. The *Relative Time Measure* is the weakest one, with accuracy of 60 % and some *off-by-2 errors* (poor performance measured as excellent). The *Execution Count Measure* and *Absolute Time Measure* achieve accuracy of about 70 % and made no *off-by-2 errors*. Although having similar accuracy, they differ significantly in the direction of the errors: the former is too benevolent, while the latter is too strict. Whether one is preferable over the other would depend on the costs associated with each type of misclassification.

DISCUSSION AND FUTURE WORK

Instead of using binary success, the currently prevalent choice of performance measure, we propose to to use a few discrete performance levels with universal interpretation, such as *failed*, *poor*, *good*, and *excellent*. We have illustrated that designing such a performance measure is nontrivial, but possible. However, the case study was limited to a specific block-based programming activity and only single-feature performance measures. For more complex programming exercises, the quality of the final program is a natural candidate for a criterion in a performance measure. However, our preliminary analysis of data from other types of introductory programming exercises (turtle graphics and Python programming with text and numbers) indicates that even in these cases most successful solutions are very similar. Nevertheless, this aspect requires further research, together with the exploration of methods for combining multiple criteria, e.g., time, number of executions, and the quality of the solution. Another possible extension is the refinement of the *failed* category (e.g., "nearly solved", "serious failed attempt", "unserious attempt").

We suggest using a performance measure with just a few parameters, which have clear interpretation and can be estimated a priori by the task authors. These parameters can then be iteratively improved once some performance data are collected. The optimization of the performance measure is complicated by its interaction with other components of an adaptive learning system. In our case study, we highlighted the importance of simultaneously improving the domain model, making sure that the tasks that share the same thresholds are indeed homogeneous with respect to their difficulty.

Other components of intelligent tutoring systems, such as the student and tutor model, also influence the collected data used to optimize and evaluate the performance measure. The learned performance measure then prepares the input data to train these models, closing the loop. Even more subtle interaction is between the performance measure and user interface: indicating what the performance measure is, e.g., by showing an execution counter, affects students' behavior and hence the collected data. Future research should shed light on the impact of these interdependencies and create a methodology for unbiased evaluation of performance measures.

REFERENCES

1. Tomáš Effenberger and Radek Pelánek. 2018. Towards making block-based programming activities adaptive. In *Proc. of Learning at Scale*. ACM, 13.
2. S Klinkenberg, M Straatemeier, and HLJ Van der Maas. 2011. Computer adaptive practice of Maths ability using a new item response model for on the fly ability and difficulty estimation. *Computers & Education* 57, 2 (2011), 1813–1824.
3. Radek Pelánek. 2017. Bayesian knowledge tracing, logistic models, and beyond: an overview of learner modeling techniques. *User Modeling and User-Adapted Interaction* 27, 3 (2017), 313–350.
4. Radek Pelánek. 2018a. The details matter: methodological nuances in the evaluation of student models. *User Modeling and User-Adapted Interaction* 28 (2018), 207–235. Issue 3.
5. Radek Pelánek. 2018b. Exploring the utility of response times and wrong answers for adaptive learning. In *Proc. Learning at Scale*. ACM, 18.
6. Jesse Schell. 2014. *The Art of Game Design: A book of lenses*. AK Peters/CRC Press.
7. Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. 2017. Learning to represent student knowledge on programming exercises using deep learning. In *Proc. of ducational Data Mining*. 324–329.
8. Yutao Wang and Neil Heffernan. 2013. Extending knowledge tracing to allow partial credit: Using continuous versus binary nodes. In *Proc. of Artificial Intelligence in Education*. Springer, 181–188.
9. Cameron Wilson. 2015. Hour of code—a record year for computer science. *ACM Inroads* 6, 1 (2015), 22–22.