Peer Tutor Matching for Introductory Programming: Data-Driven Methods to Enable New Opportunities for Help

Nicholas Diana, Carnegie Mellon University, ndiana@cmu.edu
Michael Eagle, Carnegie Mellon University, meagle@andrew.cmu.edu
John Stamper, Carnegie Mellon University, john@stamper.org
Shuchi Grover, SRI International, shuchig@cs.stanford.edu
Marie Bienkowski, SRI International, marie.bienkowski@sri.com
Satabdi Basu, SRI International, satabdi.basu@sri.com

Abstract: The number of students that can be helped in a given class period is limited by the time constraints of the class and the number of agents available for providing help. We use a classroom-replay of previously collected data to evaluate a data-driven method for increasing the number of students that can be helped. We use a machine learning model to identify students who need help in real-time, and an interaction network to group students who need similar help together using approach maps. By assigning these groups of struggling students to peer tutors (as well the instructor), we were able to more than double the number of students helped.

Introduction

While a typical classroom may be full of students experiencing the same problem and students who have solved that problem, this expertise is rarely utilized. Instead, often the only source of help is the instructor, who is most likely unable to help all the students who need help within the time constraints of the class period. To address this problem, we propose and evaluate several methods for improving the efficiency of student assistance using machine learning. Diana et al. (2017b) showed that low-level log data from the Alice introductory programming environment can be used to accurately predict student grades, and that they could increase the number of students helped by matching struggling students to a peer tutor based on the similarity of their code. A subsequent study (2017a) found that the accuracy and interpretability of the previously reported predictive model could be improved by increasing the grain size of the features from a vocabulary of terms derived through natural language processing (NLP) to small snippets of code. We explore how this improvement impacts peer tutor matching and the efficiency of providing help more generally. Also, we use an interaction network graph to test if students who may benefit from the same kind of help can be grouped together, increasing the efficiency of the instructor or peer tutor.

Methods

The data used in the current study were originally collected by Werner et al. (2012) as part of a two-year project exploring the impact of game design and programming on the development of computer science skills. The students were asked to complete an assessment task called the *Fairy Assessment*. The current experiment closely follows the data transformation methodology reported in (Diana et al., 2017b) to convert raw log data into program representations called *code-states* and the code-state complexity reduction methodology reported in (Diana et al., 2017a) to reduce code-states to smaller, *code-chunks*.

We used ridge regression to predict students' grades. We compared two methods for generating the features inputted into the regression. In the first method, features were a vocabulary of NLP terms generated from the students' code-states. In the second method, each code-state was first converted into a list of code-chunks, and then into a *chunk-frequency vector*. A chunk-frequency vector is a vector whose length is equal to the total number of features being considered in the model. Each value in the vector corresponds to the frequency of the respective code-chunk.

The predicted grades were also used to estimate which students need help and which students may be able to provide help. We call the students classified as needing help using their actual grades *low-performing students*. This classification serves as the ground-truth that we use to evaluate our predictive model. In a real-world implementation, we would not have access to the actual grades, so we must estimate them and use those estimates to classify students as needing help. If a student's predicted grade was in the bottom quartile, and they have not been helped or are not currently being helped ("helped" status persists across time), then that student was added to the group of students who still need to be helped, which we call the *Help Pool*. If a student's predicted grade was in the top quartile, and they are not currently helping a student, then that student was added to the group of students who may be able to help other students, which we call the *Tutor Pool*. For each student in the *Help Pool*, we first checked to see if the instructor was available to help. If so, the instructor was assigned to that student. If the instructor was unavailable (i.e., helping another student), then we searched for a peer tutor. We used a network graph of each code-state (or code-chunk frequencies) for each user to match tutees to tutors. We searched for tutors who shared a common ancestor node (i.e., shared a previous program state) with the tutee. These tutors were added to a pool of potential tutors. From that pool, we selected the tutor with the common ancestor node that was closest (i.e., least number of steps away) to the tutee's current node. The same method applied if segmenting was used, except that instead of matching the instructor or peer tutor to one student, the instructor or tutor was matched to a segment of students with a similar problem.

While the primary goal of our previous work (Diana et al., 2017b) was to evaluate how well our model could correctly classify students who would go on to have a low final grade (low-performing students), the primary goal of the current experiment is to evaluate how efficient this intervention would be. That is, we were interested in what percentage of those low-performing students could be helped, and how we can maximize that percentage. We call this ratio the *Efficiency Index* (EI), and define it formally as:

$$EI = \frac{Low\ Performing\ Students\ Helped/Being\ Helped}{Low\ Performing\ Students}$$

Results

We compared models using a linear mixed model with the measure of interest as the dependent variable, model as a fixed effect, and time bin as a random effect. We hypothesized that we can use low-level programming data to group similar low-performing students together so that they can be helped as a group. To test this, we first replicated our previously reported model to use as a baseline measure. Then, we generated a new model that incorporated segmenting. Both models used NLP features in a ridge regression and an interaction network graph built using code-states as nodes. We found that the EI (M=0.467, SD=0.210) of the model that incorporated segmenting was significantly higher (p<.001) than the baseline model (M=0.305, SD=0.190).

We also hypothesized that using the presence or absence of code-chunks as model features would improve the performance of the model. To test this, we generated a model using a sample of the code-chunks from our previous work that were shown to be good predictors of learning outcomes (Diana et al., 2017a). We generated a model using these 16 code-chunk features (rather than the NLP-derived terms used in the baseline model), and found that this code-chunk model had a significantly lower (p<.001) RMSE (M=0.246, SD=0.064) than the baseline model (M=0.263, SD=0.073).

Finally, we hypothesized that a network graph generated using code-chunks as nodes would lead to greater coverage and a higher EI. To test this, we generated a model using the same 16 code-chunks described above as features in the regression. A network graph was also generated to incorporate segmenting. However, instead of each node corresponding to a code-state, each node corresponded to a chunk-frequency vector. Representing nodes as chunk-frequency vectors more than doubled the coverage (coverage=0.924) compared to the network graph generated using code-states (coverage=0.374). The EI of the model using chunk-frequency vectors to generate the network graph (M=0.813, SD=0.128) also had a significantly higher (p<.001) EI than the model using code-states (M=0.428, SD=0.217).

Conclusion

In this paper, we explored a method for increasing the amount of help given in a typical class period. Our previous work demonstrated that we can use a predictive model to accurately identify students who may need help. We built off of this work in two ways. First, we improved the accuracy of the predictive model by using more relevant features. Second, we drastically increased the number of students able to be helped from, on average, 3.72 to 9.92 by grouping low-performing students together to be helped as a group (in combination with better model features). These results suggest that using low-level log data to group and match low-performing students to peer tutors may be an effective way to increase the amount of help given in a classroom.

References

Diana, N., Eagle, M., Stamper, J., Grover, S., Bienkowski, M., & Basu, S. (2017a). Data-driven generation of rubric parameters from an educational programming environment. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 10331 LNAI). http://doi.org/10.1007/978-3-319-61425-0_47

Diana, N., Grover, S., Eagle, M., Bienkowski, M., Stamper, J., & Basu, S. (2017b). An instructor dashboard for real-time analytics in interactive programming assignments. In ACM International Conference Proceeding Series (Vol. Part F1267). http://doi.org/10.1145/3027385.3027441

Werner, L., Denner, J., & Campe, S. (2012). The Fairy Performance Assessment : Measuring Computational Thinking in Middle School. Proceedings of the 43rd ACM Technical Symposium on Computer Science Education - SIGCSE ’12, 215–220. http://doi.org/10.1145/2157136.2157200