Data-driven Generation of Rubric Parameters from an Educational Programming Environment

Nicholas Diana[1], Michael Eagle[1], John Stamper[1], Shuchi Grover[2], Marie Bienkowski[2], and Satabdi Basu[2]

[1] Human-Computer Interaction Institute, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
[2] SRI International
333 Ravenswood Avenue, Menlo Park, CA 94025
{ndiana@cmu.edu,meagle@cs.cmu.edu,john@stamper.org,shuchi.grover@sri.com,
marie.bienkowski@sri.com,satabdi.basu@sri.com}

Abstract. We demonstrate that, by using a small set of hand-graded students, we can automatically generate rubric parameters with a high degree of validity, and that a predictive model incorporating these rubric parameters is more accurate than a previously reported model. We present this method as one approach to addressing the often challenging problem of grading assignments in programming environments. A classic solution is creating unit-tests that the student-generated program must pass, but the rigid, structured nature of unit-tests is suboptimal for assessing more open-ended assignments. Furthermore, the creation of unit-tests requires predicting the various ways a student might correctly solve a problem – a challenging and time-intensive process. The current study proposes an alternative, semi-automated method for generating rubric parameters using low-level data from the Alice programming environment.

Keywords: Programming, Automatic Assessment, Alice, Educational Data Mining

1 Introduction

Manually grading programming assignments is often a time-consuming and labor-intensive process. Instructors often employ Automated Assessment Tools (AATs) to increase the efficiency and consistency of the grading process. However, the rigid evaluation criteria used by most AATs are often unable to assess more open-ended programming assignments, such as those seen in the Alice programming environment [3].

One potential approach is natural language processing (NLP). Wang et al. showed that NLP could reliably detect constructs like creative problem-solving in open-ended questions [4]. As part of our previous work, we developed a predictive NLP model of students' final grades. These NLP approaches may provide a method for automating the grading of open-ended programming assignments, but they often fail to provide an interpretable justification for the automatically-generated grades. In the current study, we expand the grain-size of our features

from an NLP term to a small object that we call a code-chunk. Using this larger grain-size, we demonstrate that: 1) we improve the accuracy of our predictive models, and 2) we increase the interpretability of the key features of our model. This second result is particularly important because it allows us to compare the semantic quality of these data-driven features against the real, human-generated rubric used to generate the students' final grades.

2 Methods

Our methodology can be roughly separated into to two stages. First, we transformed the raw, low-level log data into small objects that we call *code-chunks*. Second, we tested two methods for selecting code-chunks that may be predictive of student success: seed-based selection and L1 regularization.

2.1 Data

The data used in the current study were originally collected by Werner et al. [5]. The hand-graded rubric scores serve as the ground truth that we can use to both train and evaluate our models. We used a subset of the original data (N=227), excluding students who worked on the assessment more than 5 minutes longer than the 30 minutes allotted or with missing, ambiguous, or incorrect grade or log data.

Diana et al., describes a method for transforming linear log data into hierarchical code-states [1]. These *code-states* are created for each step in a student's log file, approximating a snap-shot of the student's program at each step. The result is a list of cumulative code-states for each student that are both more readable and more amenable to analyses.

Each level of the code-state object is translated into a single-level code-chunk. Note if a parameter is equal to an array or child object, that parameter is ignored. Instead of nesting code-chunks, a new code-chunk containing the parameters of the child element is created.

2.2 Feature Selection

We compared two methods for selecting code-chunks that may be predictive of final grade: selecting code-chunks prior to the regression and selecting code-chunks (as features) within the regression.

In the first method, we selected potentially useful code-chunks by first dividing the sample of student data into two groups along a final grade threshold: high-performing students ($finalgrade \geq threshold$) and low-performing students ($finalgrade < threshold$).

Once two groups of students had been established, we compared the relative frequencies of code-chunks between groups. This was done by first generating a list of all possible code-chunks. Then we count each occurrence of each chunk for each of the two groups. Finally, we use a chi-squared test to determine if

each chunk has a significantly higher or lower frequency in the high-performing group than in the low-performing group. These significantly more or less frequent chunks serve as the features for our grade-prediction model. Finally, we generated a linear model using all features as input to serve as a baseline.

In the second method, we used L1 regularization to select features via a lasso regression. The lasso reduces features by encouraging weights to shrink to zero. Features with a weight of zero are effectively dropped from the model, reducing the number of features [2].

2.3 Model Parameters and Cross-Validation

Unless otherwise stated, all models were generated by using 20% of the data for training the model and 80% of the data for testing the model. While the models generally perform better with a larger training set, the purpose of this paper is to provide a method for reducing instructor work. As such, limiting our training set to 20% (approximately 45 students) provides some external validity for the results we report.

We compare each of these models using Root Mean Square Error (RMSE). Each reported RMSE value is the standardized average of a Stratified Shuffle-Split Cross-Validation (Folds=100). Before cross-validation in the seed-based feature selection method, each student was labeled either a high or low-performing student according to their grade and the specified threshold. Then, for each fold the data were divided into roughly equal groups, preserving the ratio of high to low-performing students across groups. The python package scikit-learn was used for cross-validation, linear regression, and lasso regression.

3 Results

3.1 Linear and Lasso Regression

A linear model was generated to test the effect of organizing features as code-chunks (as opposed to the vocabulary of terms used in the previously reported NLP model). We found that the model using all code-chunks as features was more accurate (RMSE=0.266) than the previously reported NLP model (RMSE=0.384) at predicting final grades.

The features used in our seed-based feature selection method were selected by comparing the relative frequency of high-performing vs. low-performing code-chunks. We used a chi-squared test to determine if the frequencies between groups were significantly different ($p < .05$). On average, a very small percentage (0.016%) of code-chunks met this criteria for each fold.

Several linear models were generated to test the effect of our seed-based feature selection approach at different final grade thresholds. We explored the range of final grades from 20-30 (66-100%) as final grade thresholds and generated a linear model for each value in that range. A final grade threshold of 30 had the lowest score (RMSE=0.273) while a threshold of 26 had the highest score (RMSE=0.331).

A lasso regression model ($\alpha = 0.25$) was generated to test the effect of using L1 regularization to select features (code-chunks) rather than selecting them using the seed-based, chunk frequency method described above. We found that the lasso regression model was more accurate (RMSE=0.235) than both a linear model using the same input features (all code-chunks) (RMSE=0.266) and a linear model using the pre-selected seed-based features (RMSE=0.273).

On average the lasso regression (M=14.45, SD=2.87) selected significantly more features ($p < .001$) than the frequency-based feature selection method (M=10.12, SD=1.90). There was also a moderate correlation ($r^2 = 0.686$) between the weights of features shared by both models.

4 Discussion and Conclusion

We found that both methods of feature selection produced models that were more accurate than the previously reported predictive model. This suggests that the increased context provided by the larger grain-size of a code-chunk results in better features and a more predictive model, supporting our first hypothesis.

With respect to our second hypothesis, that increasing the granularity of features will increase the interpretability of the model, we found that several highly-weighted code-chunks present in both feature selection methods shared a resemblance to the human-generated rubric parameters.

By transforming low-level log data from a programming environment into context rich code-chunks, we were able to: 1) increase the accuracy of our predictive model (with respect to a previously reported model that used smaller-grained, NLP terms as features), and 2) draw comparisons between our data-driven rubric parameters and human-generated rubric parameters.

References

1. N. Diana, M. Eagle, J. Stamper, S. Grover, M. Bienkowski, and B. Satabdi. An instructor dashboard for real-time analytics in interactive programming assignments. 2016.
2. J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.
3. R. Pausch, T. Burnette, A. C. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. A brief architectural overview of alice, a rapid prototyping system for virtual reality, 1995.
4. H. C. Wang, C. Y. Chang, and T. Y. Li. Assessing creative problem-solving with automated text grading. *Computers and Education*, 51(4):1450–1466, 2008.
5. L. Werner, J. Denner, and S. Campe. The Fairy Performance Assessment : Measuring Computational Thinking in Middle School. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education - SIGCSE '12*, pages 215–220, 2012.