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

Nicholas Diana
Carnegie Mellon University
Pittsburgh, PA
ndiana@cmu.edu

Michael Eagle
Carnegie Mellon University
Pittsburgh, PA
meagle@cs.cmu.edu

John Stamper
Carnegie Mellon University
Pittsburgh, PA
john@stamper.org

Shuchi Grover
SRI International
Menlo Park, CA
shuchi.grover@sri.com

Marie Bienkowski
SRI International
Menlo Park, CA
marie.bienkowski@sri.com

Satabdi Basu
SRI International
Menlo Park, CA
satabdi.basu@sri.com

ABSTRACT

We demonstrate that, by using a small set of hand-graded student work, we can automatically generate rubric criteria with a high degree of validity, and that a predictive model incorporating these rubric criteria 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 the more open-ended assignments students encounter in introductory programming environments like Alice. 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 criteria using low-level data from the Alice programming environment.

CCS CONCEPTS

• **Applied computing → Computer-assisted instruction**; *Computer-managed instruction*;

KEYWORDS

Computer Science Education, Programming Education, Data-driven, Assessment

ACM Reference format:
Nicholas Diana, Michael Eagle, John Stamper, Shuchi Grover, Marie Bienkowski, and Satabdi Basu. 2018. Data-driven Generation of Rubric Criteria from an Educational Programming Environment. In *Proceedings of International Conference on Learning Analytics and Knowledge, Sydney, NSW, Australia, March 7–9, 2018 (LAK '18)*, 5 pages.
https://doi.org/10.1145/3170358.3170399

LAK '18, March 7–9, 2018, Sydney, NSW, Australia

ACM ISBN 978-1-4503-6400-3/18/03…$15.00
https://doi.org/10.1145/3170358.3170399

1 INTRODUCTION

Manually grading programming assignments is often a time-consuming and labor-intensive process. As a result, 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 [7].

The structured nature of programming data has inspired many solutions to the automatic grading problem [2]. Most automated grading systems for programming assignments operate, at a basic level, in a similar way. The system subjects the student's submitted program to a series of test cases that supply various inputs to the program. The system then monitors the output for errors or unexpected values, and grades the student's work accordingly. These solutions not only reduce the instructor's workload, but have been shown to have a positive impact on student learning [8] as well.

Unfortunately, this paradigm is often inappropriate or ineffective at grading assignments in open-ended environments such as Alice. While these environments share the capacity to produce large amounts of transactional data on student problem-solving behavior, leveraging that data in an data-driven grading system may require a different approach.

One potential solution is using natural language processing (NLP) to look for meaningful patterns in user-generated data. Wang et al. showed that NLP could reliably detect constructs like creative problem-solving in open-ended questions [10]. In the programming domain, Zen et al. demonstrated that Latent Semantic Analysis (LSA) can be used to automate grading [12]. Similarly, previous work has shown that even a less sophisticated NLP method (bag-of-words) could be used to predict student grades from low-level log data [4].

While these NLP approaches may provide a method for assisting in the grading of open-ended programming assignments, they often fail to provide an interpretable justification for the machine-generated grades. In this study, we build off of previous work to generate a model that can both predict grades in an open-ended environment, and provide an interpretable justification for those grades by indicating the presence or absence of data-driven rubric criteria. We compare two methods of selecting useful rubric criteria: L1 regularization and expert-seeding.

Using experts as a resource for quality content is not entirely novel. Stamper et al., found that using a small amount of sample data generated from experts solving the educational tasks was able to greatly reduce the amount of data needed to automatically generate feedback [9]. Rather than produce additional data, as well as to avoid potential bias in expert solutions, Mostafavi et al., successfully used a subset of "exemplar" students, those who performed well above average, to train a data-driven mastery system in a problem solving tutor [6]. The current study uses a similar approach, comparing pieces of code from exemplar and non-exemplar students to select useful rubric criteria.

1.1 Related Work

As part of our previous work, we developed a predictive model of students' final grades. We used NLP to generate a vocabulary of terms from *code-states* which are snapshots of the student-built programs that were reconstructed from their low-level log data. While the model was fairly accurate, the features the model identified as important for prediction were not very interpretable. 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.

The ability to use data to drive the generation of a predictive model has several benefits for both instructors and students. First, like all AATs, this system could reduce time spent grading. Second, because the model solutions are student-generated, it would reduce time spent trying to imagine all possible correct solutions (i.e., a large enough training set will likely include a large variety of potential solutions). Third, interpretable rubric criteria remove some of the mystery present in other NLP-based approaches. It is easy to imagine a situation where a student, unhappy with their machine-generated grade, demands an explanation from the instructor. The interpretable rubric criteria generated by our system allow the instructor to point to specific pieces of code when answering the student. Finally, these interpretable rubric criteria make it easier to explore the relationship between specific pieces of code in student-generated log data and higher-order computational skills.

2 METHODS

Our methodology can be roughly separated into to two stages that we will describe in detail below . 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. [11] 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*, in which they are required to fix several errors in a malfunctioning program. A key feature of this dataset is the way in which it was graded. Each student's program was hand-graded by two experimenters along a 24 point rubric. These grades 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 [4]. Briefly, the linear log data outputted by Alice's internal logging system was reverse-engineered into a hierarchical structure that is more representative of the student's actual program. These hierarchical structures, called *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.

2.2 Code Chunks

Previous work had used natural language processing techniques to extract a vocabulary of terms from student code-states. These terms served as features in linear model used to predict a student's final grade at various time points throughout the class period. While the model could successfully predict grades, the features that were shown to be most important to the prediction were largely uninterpretable. In an effort to increase interpretability, we explore a method for decomposing code-states into smaller, interpretable objects we call code-chunks.

To covert a code-state into a set of code-chunks, each level of the code-state object is translated into a single-level code-chunk. If a parameter represents a list of child nodes, that parameter is ignored. Instead, a new code-chunk is created for each of the child nodes. Additionally, overly specific parameters such as *"id"* are ignored. Including these parameters would give the code-chunk a degree of specificity that makes finding functionally identical code-chunks more difficult. See Figure 1 for an example conversion from code-state to code-chunks.

The ultimate goal of organizing code-state data this way is to find meaningful and important chunks of code that are predictive of student success. Once identified, these chunks can be used to construct a rubric used to grade new student data.

2.3 Feature Selection

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

2.3.1 Using Exemplar-Seeds and Code-Chunk Frequency to Select Features. 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 ($finalgrade < threshold$). Note that here the labels *high* and *low* are used for the sake of simplicity, and that students labeled as *low-performing* may

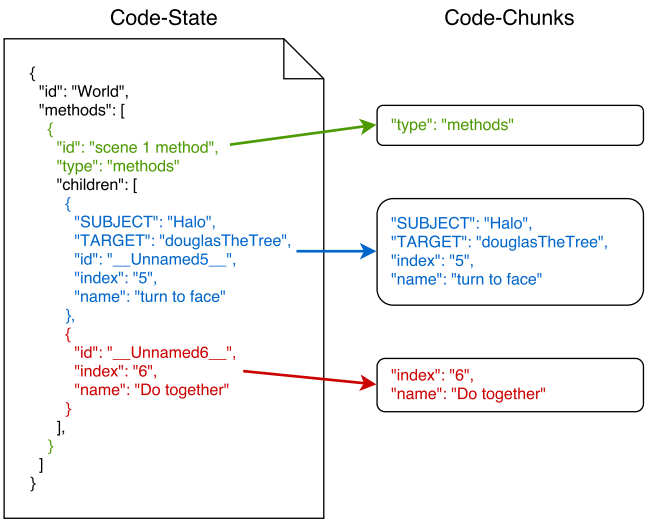

Figure 1: An example of how code-states are decomposed into code-chunks. Note that overly specific parameters like "id" are dropped. Additionally, parameters that represent lists of child nodes are also dropped. Instead, a new code-chunk is created for each of the child nodes.

have performed well on the task, but simply failed to score at or above the 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.

Note that we included code-chunks that have a significantly lower frequency in the high-performing group. Essentially, these code-chunks are markers of novice behavior that is rarely seen in students who score well. They may indicate misconceptions or *anti-patterns* of behavior [3]. Expert teachers often have an intuition about the kinds of misconceptions students may have when solving a problem, but a key contribution of this methodology is a data-driven approach for identifying those misconceptions.

To explore the space of potential features, we varied the threshold used to divide high-performing students from low-performing students. We also generated a linear model using all features as input to serve as a baseline.

2.3.2 Using Lasso Regression to Select Features. 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 [5].

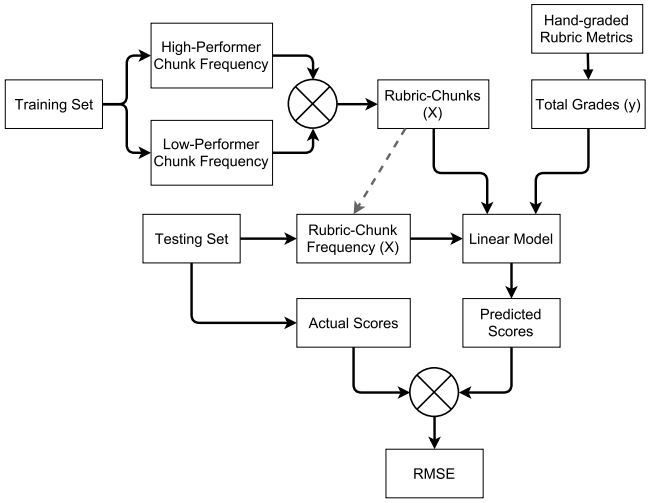

Figure 2: A visual representation of the flow of data in the seed-based feature selection method. This graph represents one fold in the cross-validation.

2.4 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 [1].

3 RESULTS

3.1 Linear Regression

3.1.1 All Code-chunks as Features. 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.

3.1.2 Seed-Based Feature Selection. 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).

3.2 Lasso Regression

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).

3.3 Comparing Selected Features

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). Figure 3 shows a moderate correlation ($r^2 = 0.686$) between the weights of features shared by both models.

In addition to comparing the features selected by each model, we were also interested in framing the features selected as rubric criteria. This allowed us to compare our data-driven feature selection against the human-generated rubric criteria used to grade the students' work. While these comparisons are inherently qualitative, there were several cases of data-driven criteria bearing considerable similarity to human-generated criteria. For example, the human-generated rubric criterion that assesses whether or not a character named *Halo* had turned some amount to face another character. The following is a strongly-weighted code-chunk shared by both models that may correspond to this criterion:

```
{
   'index': '2',
    'DIRECTION': 'left',
    'SUBJECT': 'Halo',
    'name': 'turn',
    'AMOUNT': '0.25'
}
```

Additionally, the human-generated rubric criterion that assesses whether or not a character named *LeafFlame* is resized to the value 2. Again, we find a strongly-weighted code-chunk shared by both models that may correspond to this criterion:

```
{
   'index': '0',
    'AMOUNT': '2',
    'name': 'resize',
    'SUBJECT': 'LeafFlame'
}
```

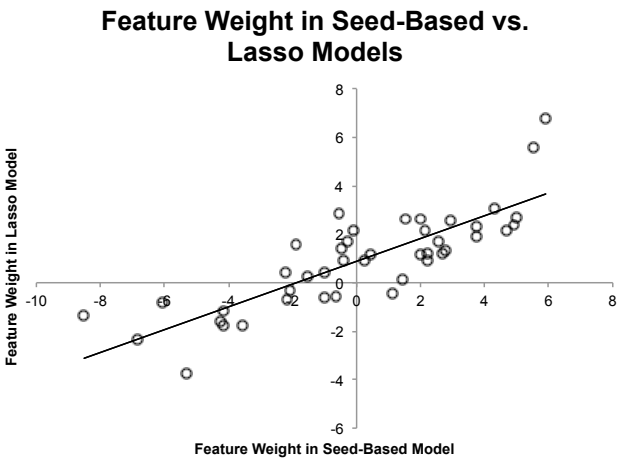

Figure 3: The weights of the features shared by each feature-selection method. We see a moderate correlation between the feature weight in the seed-based model and the feature weight in the lasso-based model.

4 DISCUSSION

4.1 Model Comparison

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.

4.2 Validity of Generated Rubric Parameters

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 criteria. This increased interpretability allows us to frame the features of our predictive model as rubric criteria. Furthermore, the accuracy of the predictive model that utilizes these automatically generated rubric parameters demonstrates the ability of a data-driven approach to reliably supplement human graders.

It is also worth noting that the human-generated rubric consisted of 24 different parameters (though many were variants of one another (e.g., criteria such as "doTogether_with_walk" and "doTogether_no_walk"). In contrast, our lasso regression selected, on average, only 14.45 features. The specificity afforded by the additional parameters present in the human-generated rubric may be key in improving the accuracy of our predictive model further.

4.3 Applications

This data-driven rubric generation method could potentially reduce instructor workload in the following ways: First, we demonstrate that a reasonably accurate model can be trained on a small fraction (20%) of the data. Second, given a small set of training data, our model selects the pieces of code from student data (code-chunks) that are predictive of final grades. Because these chunks of code are,

Table 1: Comparison of linear and lasso regression models using different feature selection methods.

Regression	Feature Selection	RMSE	Avg. # Features
Linear	None (All Code-Chunks)	0.266	NA
Linear	Frequency-Based	0.273	10.12
Lasso	L1 Regularization	0.235	14.45

in large part, interpretable, they can be framed as the rubric criteria by which the instructor can evaluate unseen student data. Finally, given that these rubric criteria were selected using a predictive model, final grades for the remaining, unseen student data can easily be predicted.

In addition to the generation of rubric criteria and grades, we believe that this approach may be used to yield more fundamental insights about the relationship between raw log data and higher-order computational skills. Using this approach and an assignment carefully designed to elicit a particular computational skills (e.g., recursion, abstraction, etc.), an instructor may be able to find the physical pieces of student code that correspond most closely with an abstract skill. Our future work will explore this possibility.

5 CONCLUSION

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. Together, these two results demonstrate a methodology that could drastically reduce the time instructor's spend grading assignments, while providing an interpretable justification for the machine-generated grades.

ACKNOWLEDGMENTS

The authors would like to thank [NAME OF GRANTS] who provided funding for this work.

REFERENCES

[1] [n. d.]. Scikit-learn: Machine Learning in Python, author=Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E., journal=Journal of Machine Learning Research, volume=12, pages=2825–2830, year=2011. ([n. d.]).

[2] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005), 83–102.

[3] Brad Appleton. 1997. Patterns and software: Essential concepts and terminology. *Object Magazine Online* 3, 5 (1997), 20–25.

[4] Nicholas Diana, Michael Eagle, John Stamper, Shuchi Grover, Marie Bienkowski, and Basu Satabdi. 2016. An Instructor Dashboard for Real-Time Analytics in Interactive Programming Assignments. (2016). https://doi.org/10.1145/3027385.3027441

[5] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The elements of statistical learning*. Vol. 1. Springer series in statistics Springer, Berlin.

[6] Behrooz Mostafavi, Michael Eagle, and Tiffany Barnes. 2015. Towards Data-driven Mastery Learning. In *Proceedings of the Fifth International Conference on Learning Analytics And Knowledge (LAK '15)*. ACM, New York, NY, USA, 270–274. https://doi.org/10.1145/2723576.2723622

[7] R. Pausch, T. Burnette, A. C. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. 1995. A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality. (1995).

[8] Raymond Scott Pettit, John D. Homer, Kayla Michelle Holcomb, Nevan Simone, and Susan A. Mengel. 2015. Are Automated Assessment Tools Helpful in Programming Courses?. In *2015 ASEE Annual Conference & Exposition*. ASEE Conferences, Seattle, Washington. https://peer.asee.org/23569.

[9] John Stamper, Tiffany Barnes, and Marvin Croy. 2011. Enhancing the automatic generation of hints with expert seeding. *International Journal of Artificial Intelligence in Education* 21, 1-2 (2011), 153–167. https://doi.org/10.3233/JAI-2011-021

[10] Hao Chuan Wang, Chun Yen Chang, and Tsai Yen Li. 2008. Assessing creative problem-solving with automated text grading. *Computers and Education* 51, 4 (2008), 1450–1466. https://doi.org/10.1016/j.compedu.2008.01.006

[11] Linda Werner, Jill Denner, and Shannon Campe. 2012. The Fairy Performance Assessment : Measuring Computational Thinking in Middle School. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education - SIGCSE '12* (2012), 215–220. https://doi.org/10.1145/2157136.2157200

[12] Kartinah Zen, D N F Awang Iskandar, and Ongkir Linang. 2011. Using latent semantic analysis for automated grading programming assignments. *2011 International Conference on Semantic Technology and Information Retrieval, {STAIR} 2011, June 28, 2011 - June 29, 2011* June (2011), 82–88. https://doi.org/10.1109/STAIR.2011.5995769