ACCE: Automatic Coding Composition Evaluator

Stephanie Rogers
UC, Berkeley
2308 Warring Street 101,
Berkeley, CA, 94704
srogers11@berkeley.edu

Steven Tang
UC, Berkeley
1888 Berkeley Way 318,
Berkeley, CA, 94704
steventang@berkeley.edu

John Canny
UC, Berkeley
637 Soda Hall, UC Berkeley
Campus, Berkeley, CA, 94704
canny@berkeley.edu

ABSTRACT

Coding style is important to teach to beginning programmers, so that bad habits don't become permanent. This is often done manually at the University level because automated Python static analyzers cannot accurately grade based on a given rubric. However, even manual analysis of coding style encounters problems, as we have seen quite a bit of inconsistency among our graders. We introduce ACCE–Automated Coding Composition Evaluator–a module that automates grading for the composition of programs. ACCE, given certain constraints, assesses the composition of a program through static analysis, conversion from code to AST, and clustering (unsupervised learning), helping automate the subjective process of grading based on style and identifying common mistakes. Further, we create visual representations of the clusters to allow readers and students understand where a submission falls, and the overall trends. We have applied this tool to CS61A–a CS1 level course at UC, Berkeley experiencing rapid growth in student enrollment–in an attempt to help expedite the involved process as well as reduce human grader inconsistencies.

Author Keywords
Composition; Clustering; Unsupervised Learning; Autograding; Assessment; Gephi; Visualization; Grading; Style; CS1; Evaluation

ACM Classification Keywords
K.3.1. Computer Uses in Education; I.5.3. Clustering

INTRODUCTION

The fact is, code is read much more often than it is written. Coding with good style results in code that is more readable, error-free, secure, extensible, and modular. Programming style has started to become a more formalized set of rules and guidelines. In practice, most style enforcement actually comes in the form of code reviews: manual analysis by experienced human readers. However, even manual code style evaluation has problems with consistency. We want students at the University level to practice good techniques while coding, to prepare them for industry or life beyond academia.

Nontrivial machine grading of student assessments is an emerging problem, as Massive Open Online Courses (MOOC) become more popular. Aimed at unlimited participation, these classes face the problem of scalability with respect to grading. Automating the manual processes of grading becomes a highly relevant approach to expanding the ability of these classes to evaluate both the learning outcomes and the quality of the assessment. ACCE–Automated Coding Composition Evaluator–is a module that attempts to automate the process of code reviews by predicting a score or feedback for the composition of computer programs through clustering. The purpose of the tool is two-fold: to provide highly detailed and targeted feedback, and to act as a verification tool–to enforce consistency between graders and for any particular grader.

RELATED WORKS

A few ways in which code is currently automatically graded include unit tests (or test-base based), static analysis checkers or feature-based approaches, satisfaction-based programs, and code coverage analyzers [1] [3] [4]. Our work is primarily based off of Huang et. al. in which they examine the syntactic and functional variability of a huge code base worth of submissions for a MOOC class [2]. In our paper, we apply their technique to a new set of data and automate the entire process.

IMPLEMENTATION

The motivating premise behind our approach is that there might only exist a limited number of common solutions that students take when solving a problem, thus feedback or grades can be extended to all solutions of the same approach. To identify different approaches, we computed the edit-distance between one project submission and all other project submissions (using the abstract syntax tree of the submission), and we repeated this computation for each submission. Submissions with low edit-distances, and similarity in structure, were clustered together.

First, the decision was made to look for common structural approaches for individual functions, rather than for entire project submissions. This was motivated by the fact that most functions in the project could be structured completely independently from one another. Each function submission was converted into its abstract syntax tree (AST), with most variables and function names anonymized. However, variable and function names that occurred as part of the project skeleton code were not anonymized, as it is structurally significant. In order to determine how structurally similar two submissions are, we calculated the *AST edit distance* pairwise between all corresponding ASTs, as described in J. Huang et. al. This pairwise computation considered the trees unordered and anonymized the names of any variables, functions or parameters, unless otherwise specified. This pairwise process is quartic in time with respect to the AST size, but only needs to be run once on a particular set of submissions. The process could likely benefit from optimizations. Lower edit distances means that submissions are more similar in structure. The inverse of an edit distance is how we calculated the *similarity score* between submissions. With similarity scores calculated pairwise for all submissions, visualizations of common structural approaches to functions could be created. The program Gephi is used, which is an interactive visualization and exploration platform for networks, complex systems, and graphs[1].

RESULTS

In Figure 1, we can see the final visualization produced by Gephi, given randomized data. Each node in the graph corresponds to an AST of an original function submission. Edges where the similarity score between two functions was below a certain threshold were not included in the graph as they should not be considered similar. Thus, an edge can be thought of as connecting two nodes that have a high enough similarity score to warrant analysis. By examining the network of solutions only in the top 5%, we observe a smaller, more manageable number of common solutions or mistakes–only a handful of clusters.

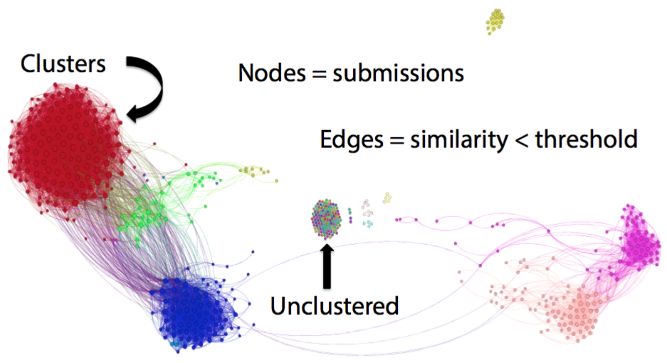

Figure 1. Cluster Visualization: Annotated Explanation

In order to produce the cluster visualization as shown in Figure 1, Gephi runs a ForceAtlas algorithm, which uses repulsion and gravity to continuously push and pull nodes away from each other. The distance between nodes and clusters directly correspond to the syntactic similarity. The clusters are colored by modularity, a measure of how well a network decomposes into modular communities. The group of multicolored nodes represents those submissions that were unclustered and therefore highly dissimilar in structure to most other submissions: somewhere around 20-25% of the submissions remained unclustered.

[1]http://gephi.org/

Identifying the need to automate the process described in the previous paragraphs, the authors created an automated tool that can accept a new code submission as a Python file to automatically produce a visualization (a .png file) with the existing clusters that were already generated. The new code submission is then emphasized in size and in color, so that the user can identify where the new submission is located. In this way, a reader can easily run the tool with a new submission and visually identify what cluster the new submission belongs to. Additionally, textual output of which cluster id the submission belongs to is output.

FUTURE WORK

With the given data from the CS61A class, we plan to perform a variety of analyses to show the effectiveness of the tool. We plan to do an in-depth manual classification of each cluster, by randomly selecting 10-15 submissions from each cluster and labeling each as "Good," "Okay," or "Poor" style. In order to provide some validation to the manual classification, we will color the nodes by the original composition scores given to each submission. Finally, we plan to manually analyze the "Poor" quality clusters to identify the common mistakes within those functions, and see how many of the nodes in that particular cluster make that same mistake. While visualizing clusters can be informative and useful, there is much future work to be done to make this process more usable to provide feedback or grades back to students. Next steps involve creating a tool that allows for clusters to be tagged with specific feedback that will be sent back to the students in a streamlined fashion. This approach is simply aimed at helping the reader grade and provide feedback, by making the process more consistent and efficient.

CONCLUSION

Automating the subjective process of grading code based off of composition is extremely difficult. We hope to leverage the common trends among submissions to make grading a more efficient process, especially when working with a dataset of such size, as MOOC courses often do. Applying feedback and scoring to submissions within the same cluster seems to be a reasonable way to grade coding submissions.

REFERENCES

1. Aggarwal V., S. S. Principles for using machine learning in the assessment of open response items: Programming assessment as a case study.
2. Huang J., Piech C., N. A. G. L. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *Proceedings of the 16th Annual Conference on Aritificial Intellgence in Education*, ACM (2013).
3. Saikkonen R., Malmi L., K. A. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ACM (2001), 133–136.
4. Singh, R., Gulwani, S., and Solar-Lezama, A. Automated semantic grading of programs. Tech. rep., MIT/Microsoft Research, 2012.