Intelligent Evaluation and Feedback in Support of a Credit-Bearing MOOC

David Joyner(✉)

Georgia Institute of Technology, Atlanta, GA 30332, USA
david.joyner@gatech.edu

Abstract. Massive Open Online Courses (MOOCs) may reach a massive number of people, but few MOOCs count for credit. Scaling rigorous assessment, feedback, and integrity checks presents difficulties. We implemented an AI system for a CS1 MOOC-for-credit to address both scale and endorsement. In this analysis, we present the design of the system and an evaluation of the course. We observe that students in the online course achieve comparable learning outcomes, report a more positive student experience, and identify AI-equipped programming problems as the primary contributor to their experiences.

Keywords: Automated evaluation · MOOCs · CS1

1 Introduction

In this work, we present an online version of a CS1 class at a large state university. The course aims to be a MOOC-for-credit: it is offered to on-campus students for degree credit, but is also available to MOOC students from all over the world for free. In order to accomplish this, mechanisms must be found for scaling up the human grading found in a traditional campus while preserving (or improving) the learning outcomes and student experience. Toward this end, the CS1 course we present here is built with a strong emphasis on applying AI to the evaluation and feedback process. At time of writing, 440 students have enrolled in the course for credit, while 12,000 students have participated in the free massive open online course (MOOC) on edX.

2 Related Work

Work has been done on applying AI to computer science evaluations. There are many historical (Web-CAT [4], OKPy [14], and Autolab [17]) and modern (Vocareum [9], Zybooks, Coursera [1], Udacity [7]) platforms and frameworks targeting this space. Wilcox [16] gives an overview of the automated evaluation space. These tools mostly focus on evaluation, but it is also important to provide feedback. Wiese et al. [15], for example, automatically generate style feedback for students, and Glassman et al. categorize submissions into patterns for tailored feedback [5]. There also exist several intelligent tutoring systems for computer programming [2, 3, 12, 13].

C. Penstein Rosé et al. (Eds.): AIED 2018, LNAI 10948, pp. 166–170, 2018.
https://doi.org/10.1007/978-3-319-93846-2_30

3 Intelligent Evaluation and Feedback

The online CS1 class described herein can be taken by any student to fulfill their state CS requirement and routinely draws more than 1000 students per year. To scale, assignments generally need to be automatically evaluated, but to maintain credit worthiness, the evaluations must be rigorous, authentic, and comparable to traditional on-campus offerings of the course. To support this, we constructed an infrastructure for intelligent evaluation and feedback. We then populated it with 300 programming problems, including 30 proctored exam problems, leveraging this framework. Additional details of the course design can be found in [8–10]; this analysis emphasizes the design of the course's intelligent feedback.

3.1 Global and Local Infrastructure

The infrastructure for intelligent evaluation and feedback has two parts: a global, general framework for initializing automated evaluation, gathering results, and presenting results to the user; and a set of local, specific parameters targeted at individual problems. We dub the global framework "Phineas" and the local parameters "Ferb".

When a student submits a coding problem, Phineas gathers together the student's code and bundles it into an inspectable object, allowing for deep code inspection and unit testing. Phineas sends these arguments to Ferb, a set of problem-specific parameters, such as a routine for intelligently generating new unit tests, a set of forbidden or required code, and explicit requirements such as method signatures.

Local evaluation of a student's submission against Ferb's parameters runs through three stages: pre-processing, unit testing, and post-processing. This **pre-processing** covers those checks necessary for the unit tests to be inspected. This stage focuses on the presence of specific functions, objects, methods, and variables.

If those preprocessing checks fail, the student is informed of that failure immediately, and subsequent checks are not run. If the preprocessing checks pass, the code is executed for **unit tests** generated by Ferb. This generates a list of the results, with each result including the input arguments, the expected output, the actual output, and whether the result is considered a pass, fail, or warning.

After running unit tests, the code is finally checked against the **post-processing** checks. These checks are typically against data generated during runtime, such as a count of the number of loop iterations or recursive calls, and thus may only run after execution. The results from these checks are appended to the corresponding lists of passed and failed results.

These results are then returned to Phineas in the form of three lists: a list of successes, a list of failures, and a list of warnings. Phineas then performs a follow-up check on the list of failures and searches for corresponding global feedback. For example, if one of the failures references an unsupported operand TypeError, Phineas may inject general feedback on the likely causes of this error.

Then, Phineas uses these lists to write a file named full_results.txt; this file lists the failed requirements, then the warnings, and then the successes. Phineas then writes a brief result summary to provide in the console window. If there are any failures, Phineas selects the first failure to provide, closing with a pointer to the full results file.

If there are no failures, Phineas first informs students that their code passed the problem; if there are warnings, it then lets the student know that there may nonetheless be room for improvement. Once the student passes the problem, Phineas adds to their workspace one or more sample solutions written by the problem author. These sample solutions demonstrate optimal or alternate solutions for implicit feedback [6].

3.2 Problem Content

The course has over 300 coding problems, each with its a version of Ferb with parameters suited to the problem's requirements. Four problem templates are available:

- **Variable Inspection:** Direct inspection of variables and their values.
- **Console Output:** Inspection of content printed when executing student code after injecting intelligently-generated alternate values for existing variables.
- **Function Output:** Comparison of output of function calls from students' code with correct code based on intelligently-generated test cases.
- **Object Inspection:** Experimenting with the success of certain object instantiations and follow-up method calls or attribute checks.

All four problem templates come equipped with additional intelligent tolerance, such as the option to ignore minor differences in whitespace in output, to automatically perform type conversions, and to ignore rounding errors as deemed necessary.

4 Course Evaluation

In checking the credit-worthiness of the online course, we are concerned with learning outcomes and student experience. Students should learn as much in the online course as in the traditional, and the student experience should be at least comparably positive. To check this, we performed a pseudo-experiment comparing students in two semesters of this online for-credit course to the equivalent on-campus course.

4.1 Learning Outcomes

To evaluate learning outcomes, students in both the online and the traditional version of the course were given the SCS1 computer science assessment as a pre-test and post-test [11]. In this analysis, we investigated the effect that the AI-supported course may have on students of different levels of prior experience. We compared students within four specific subgroups: those who have previously completed a CS course ("Prior Expertise"), those who have previously started by failed or withdrawn from a CS course ("Prior Experience"), those who are self-taught or otherwise have some informal experience ("Informal Experience"), and those with no prior experience ("No Experience"). Table 1 shows these results.

We performed t-tests on each of these eight pairs. "Prior Experience" was statistically significant at $\alpha = 0.05$ ($t = 2.14$, $p = 0.0445$). "Prior Expertise" was statistically significant at $\alpha = 0.10$ ($t = 1.84$, $p = 0.0699$). No other differences were significant. However, these subdivisions mean that the sample sizes are small and this data has

Table 1. Pre-test and post-test scores of both course versions by prior experience. Mean test scores are bolded; sample sizes are italicized; standard deviations are unstylized.

	Prior expertise						Prior experience					
	Traditional			Online			Traditional			Online		
Pre-test	**8.49**	*75*	7.65	**10.31**	*39*	5.43	**6.33**	*18*	4.01	**6.73**	*15*	2.46
Post-test	**10.00**	*37*	5.92	**12.26**	*39*	4.74	**7.11**	*9*	5.90	**11.43**	*14*	3.84
	Informal experience						No experience					
	Traditional			Online			Traditional			Online		
Pre-test	**9.41**	*41*	5.46	**7.95**	*22*	4.16	**5.40**	*103*	2.14	**5.22**	*46*	2.27
Post-test	**10.30**	*20*	7.15	**13.00**	*20*	6.56	**8.79**	*53*	5.08	**9.33**	*46*	4.04

been re-tested, and so replication in future semesters is needed. Additionally, this analysis only evaluates starting points and outcomes, not learning gains: difference in pre-test scores suggests students with prior expertise did not learn significantly more in the online section than the traditional (+2.34 vs. +1.51), but students with informal experience may have learned far more (+5.05 in the online version, +0.89 in the traditional version). The sample sizes are too small, however, to include only students who took both tests, and so this difference may be due to a response bias.

4.2 Student Experience

We asked students in both sections to evaluate their version of the course on a 7-point Likert scale. We performed a two-tailed Mann-Whitney *U* Test on the distributions to test for differences and summarized responses with interpolated medians. Online students rated their version higher than traditional students (6.35 to 5.58) with statistical significance ($Z = -5.09$, $p < 0.01$). When asking students to compare the class to other courses, students rated the online course as comparing more favorably (6.07) than the traditional course (5.37) with statistical significance ($Z = -4.61$, $p < 0.01$).

5 Conclusions

These evaluations support this AI-based MOOC-for-credit. Compared to a traditional section, students in the online section learned as much (as assessed by pre-test and post-test scores) while also reporting a higher student satisfaction. These empirical advantages come alongside procedural and economic advantages as well. Due to the heavy emphasis on AI-based grading and feedback, no teaching assistants are required to administer the course (although support for office hours, recitations, and forums are helpful). This, in turn, allows the course to be released in a self-paced MOOC because there is no manual evaluation to mandate a human grading workflow. Finally, the persistent nature of the MOOC mean that improvements carry over automatically to future semesters: all work is an investment into all future semesters instead of only into the current semester.

References

1. Alber, S., Debiasi, L.: Automated assessment in massive open online courses. Seminar aus Informatik, University of Salzburg, July 2013
2. Brusilovsky, P., Schwarz, E., Weber, G.: ELM-ART: an intelligent tutoring system on world wide web. In: Frasson, C., Gauthier, G., Lesgold, A. (eds.) ITS 1996. LNCS, vol. 1086, pp. 261–269. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61327-7_123
3. Butz, C.J., Hua, S., Maguire, R.B.: A web-based intelligent tutoring system for computer programming. In: Web Intelligence 2004, pp. 159–165. IEEE, September 2004
4. Edwards, S.H., Perez-Quinones, M.A.: Web-CAT: automatically grading programming assignments. In: ACM SIGCSE Bulletin, vol. 40, no. 3, pp. 328–328. ACM, June 2008
5. Glassman, E.L., Scott, J., Singh, R., Guo, P.J., Miller, R.C.: OverCode: visualizing variation in student solutions to programming problems at scale. ACM Trans. Comput. Hum. Interact. (TOCHI) **22**(2), 7 (2015)
6. Goel, A., Joyner, D.A.: Formative assessment and implicit feedback in online learning. In: Proceedings of Learning with MOOCs III, Philadelphia, PA (2016)
7. Goel, A., Joyner, D.A.: Using AI to teach AI: lessons from an online AI class. AI Mag. **38** (2), 48–58 (2017)
8. Joyner, D.A.: Congruency, adaptivity, modularity, and personalization: four experiments in teaching introduction to computing. In: Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale, pp. 307–310. ACM, April 2017
9. Joyner, D.: Building purposeful online learning: outcomes from blending CS1. In: Margulieux, L., Goel, A. (eds.) Blended Learning in Practice: A Guide for Researchers and Practitioners. MIT Press (in press)
10. Joyner, D.: Towards CS1 at scale: building and testing a MOOC-for-credit candidate. In: Proceedings of the Fifth (2018) ACM Conference on Learning @ Scale. ACM, June 2018
11. Parker, M.C., Guzdial, M., Engleman, S.: Replication, validation, and use of a language independent CS1 knowledge assessment. In: Proceedings of the 2016 ACM Conference on International Computing Education Research, pp. 93–101. ACM, August 2016
12. Reiser, B.J., Anderson, J.R., Farrell, R.G.: Dynamic Student Modelling in an Intelligent tutor for LISP programming. In: IJCAI 1985, pp. 8–14, August 1985
13. Soloway, E.M., Woolf, B., Rubin, E., Barth, P.: MENO-II: an intelligent tutoring system for novice programmers. In: Proceedings of the 7th International Joint Conference on Artificial Intelligence Volume 2, pp. 975–977. Morgan Kaufmann Publishers Inc., August 1981
14. Sridhara, S., Hou, B., Lu, J., DeNero, J.: Fuzz testing projects in massive courses. In: Proceedings of the Third (2016) ACM Conference on Learning @ Scale, pp. 361–367. ACM, April 2016
15. Wiese, E.S., Yen, M., Chen, A., Santos, L.A., Fox, A.: Teaching students to recognize and implement good coding style. In: Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale, pp. 41–50. ACM, April 2017
16. Wilcox, C.: Testing strategies for the automated grading of student programs. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education, pp. 437–442. ACM, February 2016
17. Zimmerman, J. Autolab: Autograding for All. Accessed http://autolab.github.io/2015/03/autolab-autograding-for-all/