Syntactic and Functional Variability of a Million Code Submissions in a Machine Learning MOOC

Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas
Stanford University

Abstract. In the first offering of Stanford's Machine Learning Massive Open-Access Online Course (MOOC) there were over a million programming submissions to 42 assignments — a dense sampling of the range of possible solutions. In this paper we map out the syntax and functional similarity of the submissions in order to explore the variation in solutions. While there was a massive number of submissions, there is a much smaller set of unique approaches. This redundancy in student solutions can be leveraged to "force multiply" teacher feedback.

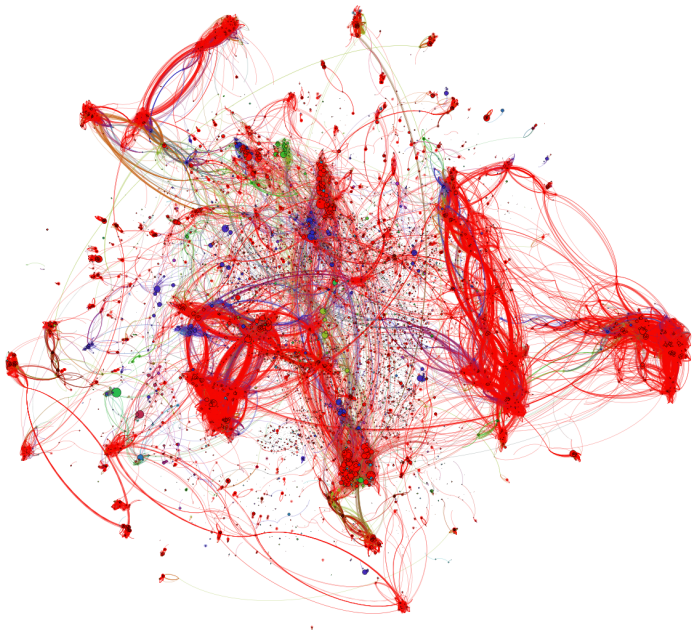


Fig. 1. The landscape of solutions for "gradient descent for linear regression" representing over 40,000 student code submissions with edges drawn between syntactically similar submissions and colors corresponding to performance on a battery of unit tests (red submissions passed all unit tests).

1 Introduction

Teachers have historically been faced with a difficult decision on how much personalized feedback to provide students on open-ended homework submissions

such as mathematical proofs, computer programs or essays. On one hand, feedback is a cornerstone of the educational experience which enables students to learn from their mistakes. On the other hand, giving comments to each student can be an overwhelming time commitment [4]. In contemporary MOOCs, characterized by enrollments of tens of thousands of students, the cost of providing informative feedback makes individual comments unfeasible.

Interestingly, a potential solution to the high cost of giving feedback in massive classes is highlighted by the volume of student work. For certain assignment types, most feedback work is redundant given sufficiently many students. For example, in an introductory programming exercise many homework submissions are similar to each other and while there may be a massive number of submissions, there is a much smaller variance in the content of those submissions. It is even possible that with enough students, the entire space of reasonable solutions is covered by a subset of student work. We believe that if we can organize the space of solutions for an assignment along underlying patterns we should be able to "force multiply" the feedback work provided by a teacher so that they can provide comments for many thousands of students with minimal effort.

Towards the goal of force multiplying teacher feedback, we explore variations in homework solutions for Stanford's Machine Learning MOOC that was taught in Fall of 2011 by Andrew Ng (ML Class), one of the first MOOCs taught. Our dataset consists of over a million student coding submissions, making it one of the largest of its kind to have been studied. By virtue of its size and the fact that it constitutes a fairly dense sampling of the possible space of solutions to homework problems, this dataset affords us a unique opportunity to study the variance of student solutions. In our research, we first separate the problem of providing feedback into two dimensions: giving output based feedback (comments on the functional result of a student's program) and syntax based feedback (comments on the stylistic structure of the student's program). We then explore the utility and limitations of a "vanilla" approach where a teacher provides feedback only on the k most common submissions. Finally we outline the potential for an algorithm which propagates feedback on the entire network of syntax and output similarities. Though we focus on the ML Class, we designed our methods to be agnostic to both programming language, and course content.

Our research builds on a rich history of work into finding similarity between programming assignments. In previous studies researchers have used program similarity metrics to identify plagiarism [1], provide suggestions to students' faced with low level programming problems [2] and finding trajectories of student solutions [3]. Though the similarity techniques that we use are rooted in previous work, the application of similarity to map out a full, massive class is novel.

2 ML Class by the numbers

When the ML Class opened in October 2011 over 120,000 students registered. Of those students 25,839 submitted at least one assignment, and 10,405 submitted solutions to all 8 homework assignments (each assignment had multiple parts

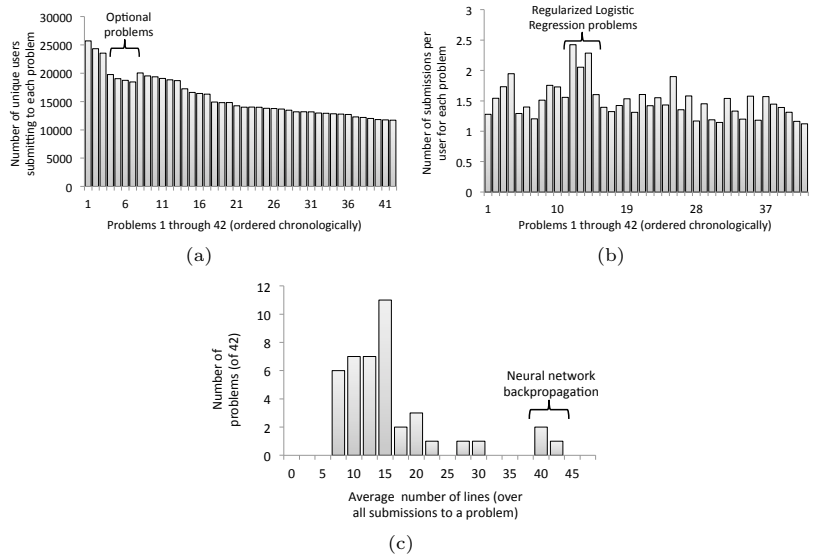


Fig. 2. (a) Number of submitting users for each problem; (b) Number of submissions per user for each problem; (c) Histogram over the 42 problems of average submission line counts.

which combined for a total of 42 coding based problems) in which students were asked to program a short matlab/octave function. These homeworks covered topics such as regression, neural networks, support vector machines, among other topics. Submissions were assessed via a battery of unit tests where the student programs were run with standard input and assessed on whether they produced the correct output. The course website provided immediate confirmation as to whether a submission was correct or not and users were able to optionally resubmit after a short time window.

Figure 2(a) plots the number of users who submitted code for each of the 42 coding problems. Similarly, Figure 2(b) plots the average number of submissions per student on each problem and reflects to some degree its difficulty.

In total there were 1,008,764 code submissions with typical submissions being quite short — on average a submission was 16.44 lines long (after removing comments and other unnecessary whitespace). Figure 2(c) plots a histogram of the average line count for each of the 42 assignments. There were three longer problems — all relating to the backpropagation algorithm for neural networks.

3 Functional variability of code submissions

First, we examine the collection of unit test outputs for each submitted assignment (which we use as a proxy for *functional variability*). In the ML Class, the

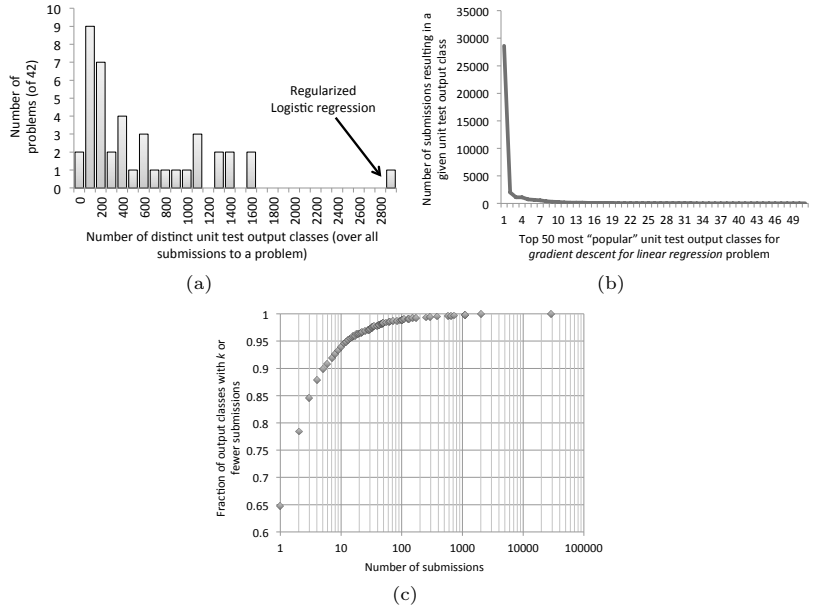


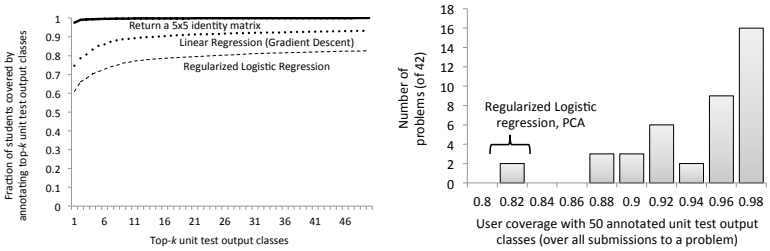
Fig. 3. (a) Histogram over the 42 problems of the number of distinct unit test outputs; (b) Number of submissions to each of the 50 most common unit test outputs for the "gradient descent for linear regression" problem; (c) Fraction of distinct unit test outputs with k or fewer submissions. For example, about 95% of unit test outputs owned fewer than 10 submissions.

unit test outputs for each program are a set of real numbers, and we consider two programs to be functionally equal if their unit test output vectors are equal.[1]

Not surprisingly in a class with tens of thousands of participants, the range of the outputs over all of the homework submissions can be quite high even in the simplest programming assignment. Figure 3(a) histograms the 42 assigned problems with respect to the number of distinct unit test outputs submitted by all students. On the low end, we observe that the 32,876 submissions to the simple problem of constructing a 5×5 identity matrix resulted in 218 distinct unit test output vectors. In some sense, the students came up with 217 wrong ways to approach the identity matrix problem. The median number of distinct outputs over all 42 problems was 423, but at the high end, we observe that the 39,421 submissions to a regularized logistic regression problem produced 2,992 distinct unit test outputs!

But were there *truly* nearly 3,000 distinct wrong ways to approach regularized logistic regression? Or were there only a handful of "typical" ways to be wrong and a large number of submissions which were each wrong in their own unique way? In the following, we say that a unit test output vector v *owns* a submission

[1] The analysis in Section 4 captures variability of programs at a more nuanced level of detail



(a) (b)

Fig. 4. (a) Number of students covered by the 50 most common unit test outputs for several representative problems; (b) Histogram over the 42 problems of number of students covered by the top 50 unit test outputs for each problem. Observe that for most problems, 50 unit test outcomes is sufficient for covering over 90% of students.

if that submission produced v when run against the given unit tests. We are interested in common or "popular" outputs vectors which own many submissions.

Figure 3(b) visualizes the popularity of the 50 unit class output vectors which owned the most submissions for the gradient descent for linear regression problem. As with all problems, the correct answer was the most popular, and in the case of linear regression, there were 28,605 submissions which passed all unit tests. Furthermore, there were only 15 additional unit test vectors which were the result of 100 submissions or more, giving some support to the idea that we can "cover" a majority of submissions simply by providing feedback based on a handful of the most popular unit test output vectors. On the other hand, if we provide feedback for only a few tens of the most popular unit test outputs, we are still orphaning in some cases thousands of submissions. Figure 3(c) plots the fraction of output vectors for the linear regression problem again which own less than k submissions (varying k on a logarithmic scale). The plot shows, for example, that approximately 95% of unit test output vectors (over $1,000$ in this case) owned 10 or fewer submissions. It would have been highly difficult to provide feedback for this 95% using the vanilla output-based feedback strategy.

To better quantify the efficacy of output-based feedback, we explore the notion of *coverage* — we want to know how many students in a MOOC we can "cover" (or provide output-based feedback for) given a fixed amount of work for the teaching staff. To study this, consider a problem P for which unit test output vectors $S = \{s_1, \ldots, s_k\}$ have been manually annotated by an instructor. This could be as simple as "good job!", to "make sure that your for-loop covers special case X". We say that a student is covered by S if every submitted solution by that student for problem P produces unit test outputs which lie in S. Figure 4(a) plots the number of students which are covered by the 50 most common unit test output vectors for several representative problems. By and large, we find that annotating the top 50 output vectors yields coverage of 90% of students or more in almost all problems (see Figure 4(b) for histogrammed output coverage over the 42 problems). However, we note that in a few cases, the top 50 output vectors might only cover slightly over 80% of students, and that even at

90% coverage, typically between 1000-2000 students are *not* covered, showing limitations of this "vanilla" approach to output-based feedback.

Thus, while output-based feedback provides us with a useful start, the vanilla approach has some limitations. More importantly however, output based feedback can often be too much of an oversimplification. For example, output-based feedback does not capture the fact that multiple output vectors can result from similar misconceptions and conversely that different misconceptions can result in the same unit test outputs. Success of output-based feedback depends greatly on a well designed battery of unit tests. Moreover, coding style which is a critical component of programming cannot be captured at all by unit test based approaches to providing feedback. In the next sections, we discuss a deeper analysis which delves further into program structure and is capable of distinguishing the more stylistic elements of a submission.

4 Syntactic variability of code submissions

In addition to providing feedback on the functional output of a student's program, we also investigate our ability to give feedback on programming style. The syntax of code submission in its raw form is a string of characters. While this representation is compact, it does not emphasize the meaning of the code. To more accurately capture the structure of a programming assignment, we compare the corresponding Abstract Syntax Tree (AST) representation.

This task is far more difficult due to the open ended nature of programming assignments which allows for a large space of programs. There were over half a million unique ASTs in our dataset. Figure 5(b) shows that homework assignments had substantially higher syntactic variability than functional variability. Even if a human labeled the thirty most common syntax trees for the Gradient Descent part of the Linear Regression homework, the teacher annotations would cover under 16% of the students. However, syntactic similarity goes beyond binary labels of "same" or "different". Instead, by calculating the *tree edit distance* between two ASTs we can measure the degree to which two code submissions are similar. Though it is computationally expensive to calculate the similarity between all pairs of solutions in a massive class, the task is feasible given the dynamic programming edit distance algorithm presented by Shasha et al [5] . While the algorithm is quartic in the worst case, it is quadratic in practice for student submission. By exploiting the [5] algorithm and using a computing cluster, we are able to match submissions at MOOC scales.

By examining the network of solutions within a cutoff edit distance of 5, we observe a smaller, more manageable number of common solutions. Figure 1 visualizes this network or landscape of solutions for the linear regression (with gradient descent) problem, with node representing a distinct AST and node sizes scaling logarithmically with respect to the number of submissions owned by that AST. By organizing the space of solutions via this network, we are able to see clusters of submissions that are syntactically similar, and feedback for one AST could potentially be propagated to other ASTs within the same cluster.

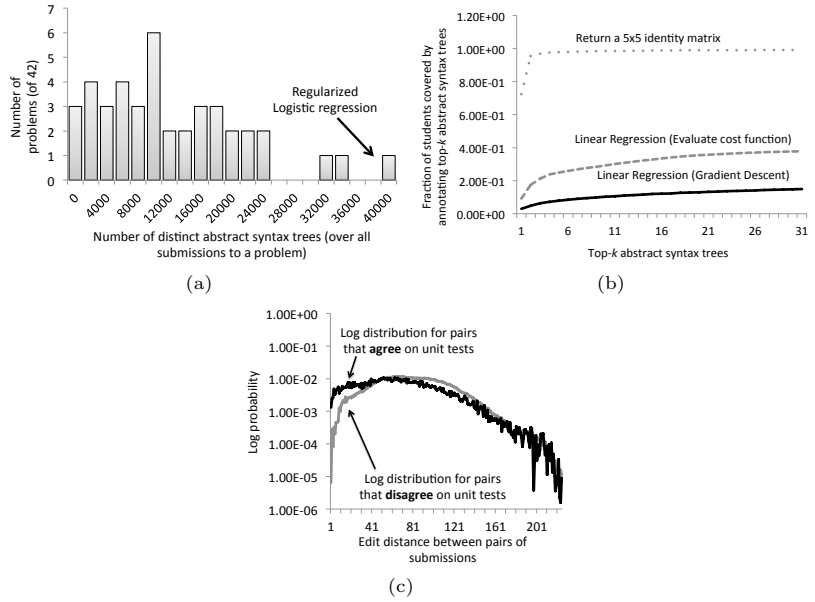


Fig. 5. (a) Histogram of the number of distinct abstract syntax trees (ASTs) submitted to each problem.; (b) Number of students covered by the 30 most common ASTs for several representative problems; (c) (Log) distribution over distances between pairs of submissions for pairs who agree on unit test outputs, and pairs who disagree. For very small edit distances (<10 edits), we see that the corresponding submissions are typically also functionally similar (i.e., agree on unit test outputs).

Figure 1 also encodes the unit test outputs for each node using colors to distinguish between distinct unit test outcomes.[2] Note that visually, submissions belonging to the same cluster typically also behave similarly in a functional sense, but not always. We quantify this interaction between functional and syntactic similarity in Figure 5(c) which visualizes (log) distributions over edit distances between pairs of submissions who *agree* on unit test outcomes and pairs of submissions who *disagree* on unit test outcomes. Figure 5(c) shows that when two ASTs are within approximately 10 edits from each other, there is a high probability that they are also functionally similar. Beyond this point, the two distributions are not significantly different, bearing witness to the fact that programs that behave similarly can be implemented in significantly different ways.

5 Discussion and ongoing work

The feedback algorithm outlined in this paper lightly touches on the potential for finding patterns that can be utilized to force multiply teacher feedback. One

[2] Edge colors are set to be the average color of the two endpoints.

clear path forward is to propagate feedback, not just for entire programs, but also for program parts. If two programs are different yet share a substantial portion in common we should be able to leverage that partial similarity.

Though we focused our research on creating an algorithm to semi-automate teacher feedback in a MOOC environment, learning the underlying organization of assignment solutions for an entire class has benefits that go beyond those initial objectives. Knowing the space of solutions and how students are distributed over that space is valuable to teaching staff who could benefit from a more nuanced understanding of the state of their class. Moreover, though this study is framed in the context of MOOCs, the ability to find patterns in student submissions should be applicable to any class with a large enough corpus of student solutions, for example, brick and mortar classes which give the same homeworks over multiple offerings, or Advanced Placement exams where thousands of students answer the same problem.

References

1. D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. In *ACM SIGCSE Bulletin*, volume 31, pages 266–270. ACM, 1999.
2. B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the 28th international conference on Human factors in computing systems*, pages 1019–1028. ACM, 2010.
3. C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 153–160. ACM, 2012.
4. P. M. Sadler and E. Good. The impact of self-and peer-grading on student learning. *Educational assessment*, 11(1):1–31, 2006.
5. D. Shasha, J. T.-L. Wang, K. Zhang, and F. Y. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):668–678, 1994.