Codewebs: Scalable Homework Search for Massive Open Online Programming Courses

Andy Nguyen
Stanford University

Christopher Piech
Stanford University

Jonathan Huang
Stanford University

Leonidas Guibas
Stanford University

ABSTRACT

Massive open online courses (MOOCs), one of the latest internet revolutions have engendered hope that constant iterative improvement and economies of scale may cure the "cost disease" of higher education. While scalable in many ways, providing feedback for homework submissions (particularly open-ended ones) remains a challenge in the online classroom. In courses where the student-teacher ratio can be ten thousand to one or worse, it is impossible for instructors to personally give feedback to students or to understand the multitude of student approaches and pitfalls. Organizing and making sense of massive collections of homework solutions is thus a critical web problem. Despite the challenges, the dense solution space sampling in highly structured homeworks for some MOOCs suggests an elegant solution to providing quality feedback to students on a massive scale.

We outline a method for decomposing online homework submissions into a vocabulary of "code phrases", and based on this vocabulary, we architect a queryable index that allows for fast searches into the massive dataset of student homework submissions. To demonstrate the utility of our homework search engine we index over a million code submissions from users worldwide in Stanford's Machine Learning MOOC and (a) semi-automatically learn shared structure amongst homework submissions and (b) generate specific feedback for student mistakes.

Codewebs is a tool that leverages the redundancy of densely sampled, highly structured homeworks in order to force-multiply teacher effort. Giving articulate, instant feedback is a crucial component of the online learning process and thus by building a homework search engine we hope to take a step towards higher quality free education.

General Terms

Algorithms; Human Factors; Measurement; Performance

Keywords

massive open online course; MOOC; education; code search; student feedback; abstract syntax tree; AST; canonicalization; semantic equivalence; Octave; Coursera

Copyright is held by the International World Wide Web Conference Committee (IW3C2). IW3C2 reserves the right to provide a hyperlink to the author's site if the Material is used in electronic media.
WWW 2014 Seoul, Korea
ACM 978-1-4503-2744-2/14/04.
http://dx.doi.org/10.1145/2566486.2568023.

1. MASSIVE SCALE COMPUTER SCIENCE EDUCATION

In the last few years, MOOCs have begun offering educational opportunities at zero cost to anyone with an internet connection [16]. People of all ages from all over the world are increasingly turning to online resources to learn, and as a result, the sheer amount of data collected about student learning has grown astronomically, eclipsing previous records. The growth of free access education and the increasing demand for higher education, especially in Science, Technology, Engineering and Math, motivates the need, more than ever, to organize the enormous amount of collected student data in an intelligent way so that we can search the data and find patterns in order to benefit both students and instructors.

Search engines for student submitted content. With the amount of collected student content increasing at a growing rate and the lines between formal classroom learning and informal online learning quickly blurring, search engines for student submitted content seem inevitable, and potentially enormously valuable. These search engines would allow one to explore a massive number of homework submissions by efficiently retrieving submissions matching specific criteria. Students would then benefit by being able to look for other students who think alike, to see how students with different viewpoints think, to find alternative strategies of approaching a homework problem, or simply to browse other work for inspiration. Instructors, on the other hand, would also benefit by being able to query a search engine in order to look for submissions of a particular type or even to count the number of students who submitted the same or a similar class of solutions.

While search engines for code have existed for a number of years (see [17, 24, 11, 12]), the nature of MOOC datasets makes for a setting that is quite distinct from that of typical code search. In the MOOC setting there is significantly more structure, since (1) submissions often take the form of single unit-testable functions written in a single language, and (2) all students submit attempts of the same problem at the same time. Due to this similarity between many submissions, it is possible to exploit the shared structure of multiple submissions in several ways. For example, this shared structure allows one to reason about the semantics of code and the intentions of the programmer.

The *Codewebs engine* that we propose in this paper is built around a massive index over the existing student code submissions which can be searched using a variety of queries and

supports a number of possible applications such as (but not limited to) finding alternative implementations, mining common approaches and misconceptions, and providing personalized feedback. Codewebs is designed to be highly scalable, allowing us to find and leverage shared structure amongst submitted programs and is able to reason about semantically similar student code. The main power of our system stems from its ability to tailor the way it deals with code semantics to each homework problem individually. Specifically, we utilize existing data to learn about each homework problem rather than requiring an excessive amount of human effort for each problem. Finally, the Codewebs system is designed so as to be easily portable to diverse programming assignments and to different programming languages remaining cognizant to the fact that instructor time is a limited resource.

Delivering scalable human feedback. Instructor time is also a limiting factor in delivering feedback to students, which is critical for learning. With the number of engaged students in a MOOC commonly exceeding ten thousand, many benefits of face-to-face instruction are easily lost. Even despite the fact that programs can be checked for correctness (i.e., unit-tested) automatically, coding assignments are complex in that they can be approached with a multitude of algorithms or strategies and typically require creativity. Thus in many coding problems, students can benefit from more feedback than a binary "correct vs. incorrect". It is difficult, however, to gauge student understanding without being able to personally peruse through tens of thousands of code submissions. And it is consequently difficult to grade student work, to provide detailed student feedback, or respond to individual questions.

Detailed assignment feedback is in general neither easily automated or even appropriate for typical crowd-sourcing platforms such as Mechanical Turk since graders are required to have a certain level of expertise. One proposed solution (pursued, for example, by the Coursera platform) has been to use peer grading instead, allowing students to grade each other. While peer grading has shown signs of initial success, it also suffers from a number of limitations including inconsistencies and biases amongst peer graders as well as sometimes an unwillingness or inability to provide high quality and constructive feedback [18].

Our method for providing scalable human feedback derives from the observation that even though there may be many thousands of unique submissions to an open-ended programming assignment the diversity reflects a much smaller number of compounded student-decision points. By recognizing "shared parts" of student solutions and decoupling different decisions, instructor feedback can be force multiplied. The idea behind our tool is to have an instructor provide detailed feedback only on a handful of submissions which we can intelligently propagate to new solutions via underlying assignment structure learned using our homework search engine. We aim to create a system where a teacher can provide detailed feedback for thousands of students with the same effort that she would spend in an ordinary college course.

The strategy of propagating teacher feedback aligns with our view that *human instructors have a key role to play in the future digital classroom*. Seamlessly incorporating human feedback into the loop bridges the gap between the typical MOOC experience and the small classroom experience. Finally, because our system discovers the commonalities amongst student submissions using the corpus of students submissions, adding more student data to our data bank will improve our statistical confidence and coverage, allowing us to provide meaningful feedback to more students, detect more classes of conceptual errors, and generally improve the utility of our system.

# submissions in total	1,008,764
# coding problems	42
Average # submissions per problem	24,018
Average # students submitting per problem	15,782
Average # of lines per submission	16.44
Average # of nodes per AST	164

Table 1: Statistics of the ML class dataset.

Overview of main contributions. To summarize, we propose in this paper a novel system, *Codewebs*, for searching through code submissions to a programming intensive MOOC as well as a methodology for scalably providing human level feedback to students in the MOOC. The main technical innovations that we have developed are as follows.

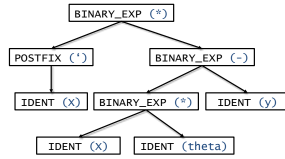
- We propose an efficient method for indexing code by "code phrases" corresponding to constituent parts of code submissions to a massive scale programming course. In particular, we introduce a novel way to query for the surroundings, or *context* of a code phrase, which allows us to reason about the behavior of code within its context.
- We develop a data driven approach for automatically finding semantically equivalent code phrases. In particular we introduce a notion of *probabilistic semantic equivalence* allowing us to connect syntax with semantics and to compare code with respect to both.
- We demonstrate applications of our search index such as bug finding (without having to execute code) and allowing a human instructor to provide feedback at MOOC scales.
- We demonstrate the scalability of the Codewebs engine and show results on code submissions to a real MOOC, Stanford's machine learning course, which had over 120,000 students enrolled worldwide, who submitted over a million code submissions in total. We also explore the variability of this dataset, showing in particular that the frequency counts of student submitted code phrases for a single programming problem follow a Zipf law distribution despite the fact that all submissions are implementations of the same function.

2. DATA FROM A PROGRAMMING INTENSIVE MOOC

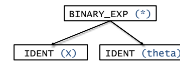
We showcase our Codewebs system using data collected from Stanford's online machine learning (ML) Class (taught by Andrew Ng), which opened in October 2011 with over 120,000 students registering. Over the duration of the course, over 25,000 students submitted at least one assignment, and 10,405 students submitted solutions to all 8 homework assignments. Each assignment had multiple parts (which combined for a total of 42 coding problems), in which students were asked to implement algorithms covered in lectures such

```
function [theta, J_history]
    = gradientDescent(X, y, theta, alpha, num_iters)

%GRADIENTDESCENT gradient descent to learn theta
%   updates theta by taking num_iters gradient
%   steps with learning rate alpha.

m = length(y); % number of training examples
J_history = zeros(num_iters, 1);

for iter = 1:num_iters
    theta = theta-alpha*1/m*(X'*(X*theta-y));
    J_history(iter) = computeCost(X, y, theta);
end
```

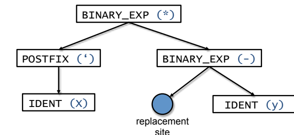
(a)



(b)



(c)



(d)

Figure 1: (a) Example code submission to the "Gradient descent (for linear regression)" problem; (b) Example abstract syntax tree for linear regression gradient expression: $X' * (X * \theta - y)$; (c) Example subtree from the AST from Figure 1(b); (d) Context of the subtree with respect to the same AST. Note the additional node denoting the "replacement site" of the context.

as regression, neural networks and support vector machines. Code for the ML class was predominantly written in Octave, a high level interpreted language similar to MATLAB, and the submissions collectively covered nearly all of Octave's basic language features. Submissions to the course website 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. We remark that high level languages such as Octave are fairly popular for instruction because they allow a student to ignore many of the low level frustrations associated with programming and concentrate on the more important higher level concepts. But they are also harder languages to automatically analyze due to their more flexible syntax and dynamic typing. As we show in this paper, data driven methods give us a way to effectively analyze submissions in such languages.

Figure 1(a) shows a correct attempt at the "Gradient descent (for linear regression)" problem assigned in the first homework. In this problem, students were asked to find a linear predictor of the output vector `y` based on the input matrix `X`. Throughout the paper, we will use this linear regression coding problem as a running example.

The Codewebs engine allows us to efficiently query this massive collection of code submissions from Stanford's ML class, all of which attempt to implement the same functionality. With so many submissions of the same programming problem, we are able to obtain a dense sampling of the solution space, in which we observe almost every conceivable way of approaching the problem both correctly and incorrectly. The dataset is thus perfect for highlighting the advantages of Codewebs which is able to exploit large amounts of data to do things that would be much more difficult to accomplish otherwise. At the same time, a massive course such as ML class can greatly benefit from a detailed student feedback tool, making our dataset also ideal for demonstrating the applicability of Codewebs. More statistics of our dataset are summarized in Table 1.

Structured representations of syntax. In addition to a string representation of each student's code submission, we parse each student submission into an *abstract syntax tree (AST)* representation. Each AST (which is equivalent to that created internally by the Octave software [3]) is stored explicitly in our database in JSON format. An example AST is depicted in Figure 1(b). Running on the ML class dataset, our parser accepts over 99.5% of submitted code, failing in a small percentage of cases (often due to submissions in different languages such as Python or MATLAB), which we discard from the dataset. Nodes in our ASTs are specified by a *type* (e.g., STATEMENT, ASSIGN) and an optional *name* (e.g., `X`, `theta`, `gradient`). Other than the statement and identifier nodes, we do not assume any specific knowledge of the semantic meanings of node types.

Reasoning with abstract syntax trees instead of the original source code allows the Codewebs system to effectively ignore cosmetic idiosyncrasies in whitespace, comments, and to some extent, differences in variable names. We find in many cases that multiple submissions that are distinct as code strings can correspond to the same AST. Thus after parsing, we retain just over half a million distinct ASTs from the original million submissions along with the number of occurrences of each unique AST.

3. EFFICIENT INDEXING OF CODE SUBMISSIONS

What basic queries or items should a code search engine index? If we viewed code simply as a string, it would be reasonable to query by terms, or phrases or regular expressions, but with the additional tree structure of our ASTs, we can go beyond traditional "flat" queries. The Codewebs engine accepts basic queries in the form of what we call *code phrases*, subgraphs of an AST which take three forms: subtrees, subforests, and contexts, which we now define. In the following, consider any AST denoted by $\mathcal{A}$:

[Subtrees] *Subtrees* represent the most basic form of code phrases and are specified by a single node $\mathbf{a}$ in AST $\mathcal{A}$ and contain all descendants of $\mathbf{a}$. If an AST $\mathcal{B}$ is a subtree of AST $\mathcal{A}$, we write $\mathcal{B} \leq \mathcal{A}$. When referring to a specific subtree rooted at a node $\mathbf{a}$ of an AST, we write $\mathcal{A}[\mathbf{a}]$.

[Subforests] In addition to subtrees, we consider *subforests* which capture everything in a contiguous sequence of statements. Specifically, we define a subforest to be a consecutive sequence of *statement subtrees* (i.e., subtrees rooted at STATEMENT nodes).

[Context] Because the function of a code snippet generally depends strongly on the surrounding region of code in which it appears, we also allow for these surroundings to be directly queried by our engine. Formally, we define the *context* of a subtree $\mathcal{A}[\mathbf{a}]$ within a larger subtree $\mathcal{A}[\mathbf{a}']$ of the same AST to be the subgraph of $\mathcal{A}[\mathbf{a}']$ which does not contain anything in $\mathcal{A}[\mathbf{a}]$ to which we add an additional leaf attached to the parent of $\mathbf{a}$, representing the "replacement site" where other subtrees could potentially be swapped in. For example, the context of the body of a for-loop with respect to the subtree rooted at that for-loop contains a variable and expression specifying termination conditions. We denote the context

by $\mathcal{A}[\mathbf{a}']\backslash\mathcal{A}[\mathbf{a}]$. Figure 1(c) shows an example subtree with its corresponding context (Figure 1(d)). We also refer to two special types of contexts:

- Given AST $\mathcal{A}$ and subtree $\mathcal{A}[\mathbf{a}]$, the *global context* of $\mathcal{A}[\mathbf{a}]$ with respect to $\mathcal{A}$ (written $\mathcal{A}\backslash\mathcal{A}[\mathbf{a}]$) refers to the tree obtained by removing $\mathcal{A}[\mathbf{a}]$ from $\mathcal{A}$.
- Given a subtree $\mathcal{A}[\mathbf{a}]$ rooted at node $\mathbf{a}$ in AST $\mathcal{A}$, the *local context* of $\mathcal{A}[\mathbf{a}]$ with respect to $\mathcal{A}$ (written $\mathcal{A}[p[\mathbf{a}]]\backslash\mathcal{A}[\mathbf{a}]$) is the context of $\mathcal{A}[\mathbf{a}]$ within the subtree rooted at $\mathbf{a}$'s parent.

Building an inverted index. We now describe the construction of the inverted index at the heart of the Codewebs system, associating possible code phrases of the above three types to lists of ASTs in the database which contain those phrases. For simplicity, we only consider *exact* code phrase matches in this section and defer the general case to Section 4. While traditional search engines [27] typically do not index arbitrarily long phrases, it makes sense in the student code setting to index with respect to *all* possible code phrases defined above since (1) student code submissions are typically limited in length (see Table 1), and since (2) virtually all of the submissions attempt to implement the same functionality, resulting in many ASTs sharing large code phrases.

The inverted index is incrementally constructed by adding one AST at a time as follows. We first preprocess all ASTs by anonymizing identifiers that are not recognized as reserved Octave identifiers or as those belonging to the provided starter code for the assignment. Then for each AST $\mathcal{A}$ in the data, we extract all relevant code phrases by iterating over all subtrees, all consecutive sequences of statements, and their respective global contexts. For each code phrase, we append $\mathcal{A}$ to its corresponding list in the inverted index (or start a new list containing $\mathcal{A}$ if the code phrase was not already contained in the index). Since we consider all possible code phrases and allow them to be arbitrarily large in size, naïvely building the index would be computationally expensive. In the following, we describe a scalable index construction algorithm based on an efficient hashing approach.

3.1 Efficient matching

Our inverted index is implemented as a hash table. To efficiently query the index, we compute hash codes for each code phrase by hashing the list obtained via a postorder traversal of its nodes. Specifically, given an AST $\mathcal{A}$ with nodes $\mathbf{a}_0, \ldots, \mathbf{a}_{m-1}$ (the postorder of $\mathcal{A}$), we hash $\mathcal{A}$ via the function: $H(\mathcal{A}) = p^m + \sum_{i=0}^{m-1} p^{m-i-1} h(\mathbf{a}_i)$, where p is a prime number and $h(\mathbf{a})$ can be any hash function encoding the type and name of the node $\mathbf{a}$. Code phrases (whose nodes can similarly be postordered) are hashed using the same function. We note that phrases that share the same postorder (despite being structurally distinct) are guaranteed to collide under H, but in practice, we find that it works sufficiently well (perhaps because it is unlikely for two distinct ASTs to simultaneously meet the constraints of sharing a postorder traversal and corresponding to valid source code).

Efficient precomputation of hashes. By exploiting the particular structure of the hash function H, we can efficiently precompute hash codes for all code phrases within an AST. We explicitly maintain a serialized representation of each AST $\mathcal{A}$ in our dataset as a flat array $L_{\mathcal{A}} = [[\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_m]]$ where the ith entry of L corresponds to the ith node ($\mathbf{a}_i$) of the postorder of $\mathcal{A}$. We record node type and name in each entry ℓ_i as well as the size (i.e., number of nodes) of the subtree $\mathcal{A}[\mathbf{a}_i]$ (which we denote by $|\mathcal{A}[\mathbf{a}_i]|$). Note that with size information, the original AST is uniquely determined by the flat array of nodes (and it is thus possible to check for *exact* matches between ASTs), but we ignore sizes during hashing.

The advantage of representing an AST $\mathcal{A}$ as a serialized list $L_{\mathcal{A}}$ is that each of the code phrases appearing in $\mathcal{A}$ can be similarly represented, and in fact, *directly constructed as a contiguous subsequence* of entries of $L_{\mathcal{A}}$. In particular, the subtree $\mathcal{A}[\mathbf{a}_i]$ corresponds precisely to the nodes in the contiguous subsequence: $[[\ell_{i-|\mathcal{A}[\mathbf{a}_i]|+1}, \ldots, \ell_i]]$. Serialized representations of subforests similarly correspond to contiguous subsequences of $L_{\mathcal{A}}$. Obtaining a serialized representation of a context is only slightly more complicated. Given the context $\mathcal{A}[\mathbf{a}']\backslash\mathcal{A}[\mathbf{a}]$, we take the subsequence of $L_{\mathcal{A}}$ corresponding to the larger subtree $\mathcal{A}[\mathbf{a}']$ but replace the entries corresponding to the smaller subtree $\mathcal{A}[\mathbf{a}]$ with a single entry denoting the replacement site node.

Together with the simple additive structure of our hash function H, the fact that all code phrases can be read off as contiguous subsequences, lends itself to a simple dynamic programming approach for precomputing all code phrase hashes of an AST in a single pass. Specifically, if we store the hashes of all prefixes of the postorder list (as well as all relevant powers of the prime used in the hash), we can compute the hash of any sublist in constant time, with:

$$H([[\ell_i, \ldots, \ell_j]]) = H([[\ell_0, \ldots, \ell_j]]) - p^{j-i+1}(H([[\ell_0, \ldots, \ell_{i-1}]]) - 1).$$

Since every code phrase contains at most two sublists from the postorder of the original AST, the hash for each code phrase can be computed in constant time using $O(m)$ precomputation time and additional storage. The same reasoning allows us to represent each code phrase as a view into subregions of the original AST. Thus we only require a constant amount of additional storage per code phrase.

4. DATA DRIVEN DISCOVERY OF SEMANTIC EQUIVALENCE CLASSES

One of the main difficulties in matching source code is that there are always many syntactically distinct ways of implementing the same functionality. For example, where one student might use a for-loop, another might equivalently use a while-loop, and if we were to require exact syntactic matches between code phrases (as was assumed in the previous section), we would not be able to capture the fact that the two submitted implementations were highly similar.

To address the issue of non-exact matches, some authors have turned to a softer notion of matching between ASTs such as tree edit distance (see [22]). But this approach, which has been used in works on generic code search as well as tutoring systems [12, 10, 21] is based only on syntax and is not able to directly capture semantic equivalence. Another more direct approach is to rely on *canonicalization* ([17, 25, 20, 21]) in which one applies semantic preserving transformations to the AST in order to reduce it to something more likely to match with other code in the database. Rivers et al. [20], for example, collapse constant functions, and use de Morgan's laws to normalize boolean expressions (among a number of other canonicalizations).

While canonicalization can be useful, it is limited by the fact that a set of semantic-preserving AST transformations

must be specified a priori. The task of designing transformation rules is highly nontrivial however, and differs for each programming language. For example, while it is reasonable to canonicalize the order of multiplication between integers in a strongly typed language such as Java, it might not be possible under a dynamically typed language such as Octave or Matlab, where the dimensions of a variable are not known until runtime. Furthermore, canonicalization cannot capture programmer intent — if a programmer incorrectly writes a code snippet, canonicalization might help to group it with other equivalently wrong snippets, but is unable to connect the the incorrect snippet with a correct counterpart.

Our approach is founded on two main observations for dealing with non-exact matches. The first observation is that in the education setting where almost all submissions attempt to implement the same functionality, it becomes feasible to design a set of semantics preserving AST transformations that is *customized* to each individual programming problem. Thus instead of having to invent a set of rules that can apply broadly to all programs, we can imagine designing specific syntax transformations that are likely to be helpful in reducing the variance of student submissions on each particular problem. In contrast with compiler optimization, canonicalization rules in the education setting do not have to be perfect — it would be acceptable to use AST transformations that are valid only for *most* students, perhaps reflecting common assumptions made about each problem. Our second observation is that this design task can be automated to a large extent, by mining a large enough dataset of existing student submissions accompanied by unit test outcomes for each submission.

EXAMPLE 1 (LENGTH(Y) AND SIZE(Y,1)). *As an illustration, consider our running example, the linear regression problem. Students were given a function prototype, assuming among other variables, that a one-dimensional response vector, y, would be passed in as input. One of the sources of variation in the problem came from the fact that* `length(y)` *and* `size(y,1)` *were both valid and common ways of referring to the number of elements in y (see for example, the first line under the function definition in Figure 1(a)). It would be wrong to call the two expressions equivalent since in general,* `length(y)` *and* `size(y,1)` *can give different results depending on the value of y. Thus, a typical canonicalization approach based on semantics preserving AST transformations would not identify these expressions as being similar. However, in the particular context of this linear regression problem, the two expressions would have been interchangeable in nearly all submissions, suggesting our approach (described next) of identifying the two expressions as being* probabilistically equivalent, *and raises the question of how to discover such identifications from data.*

There are two phases of our data driven canonicalization approach. In Section 4.1, we are given two code phrases and asked to determine whether they are semantically equivalent based on data. We then propose a simple semi-automated method in Section 4.2 that takes a human specified code phrase and searches the database for all equivalent ways of writing that code phrase.

4.1 Testing for semantic equivalence

In order to define our notion of probabilistic equivalence, we first state a formal definition of *exact* equivalence. We focus on semantics preserving AST transformations which take a subtree (or subforest) and replace it by another. A reasonable definition might then be as follows.

DEFINITION 2. *Given two subtrees (or subforests) $\mathcal{B}$ and $\mathcal{B}'$, we say that $\mathcal{B}$ and $\mathcal{B}'$ are equivalent if for every AST containing $\mathcal{B}$, replacing $\mathcal{B}$ by $\mathcal{B}'$ always yields a program that runs identically (i.e. always produces the same output given the same input).*

For brevity, we will use subtrees to refer to both subtrees and subforests in the remainder of the section. Definition 2 is a strict notion that might be useful in AST transformations for compiler optimization (for example, to replace a for-loop by an unrolled version), but as discussed above, does not capture the notion of similarity in Example 1 and is thus overly rigid for our purposes.

Probabilistic formulation of semantic equivalence. We relax two aspects of the definition of exact equivalence:

- First, instead of asking $\mathcal{B}$ and $\mathcal{B}$' to result in identical behavior under *all* inputs, we ask for identical behavior only on a collection of unit tests (which may not be exhaustive in practice), allowing us to verify similarity using data rather than having to establish formal proofs about program behavior.
- Second, instead of requiring that $\mathcal{B}$ and $\mathcal{B}$' behave similarly under all containing ASTs, we only require that the two subtrees behave similarly in the context of ASTs that have a high enough probability of being submitted to a particular problem.

Formally, let $F[\mathcal{A}]$ denote the output of an AST $\mathcal{A}$ given a battery of unit tests as input. In the following, we let P be the distribution over ASTs that could be submitted to a particular problem. We assume that P exists, but that we only have access to draws from it (the already submitted code). The hope is that reasoning with P allows us to focus our efforts on the more common solutions and to generalize about equivalences to never-before-seen ASTs. Our relaxed notion of equivalence is as follows:

DEFINITION 3. *We say that the AST subtrees $\mathcal{B}$ and $\mathcal{B}$' are α-equivalent if, whenever $\mathcal{A} \sim P$, $\mathcal{A}' \sim P$, the following condition is satisfied:*

$$P(F[\mathcal{A}] = F[\mathcal{A}'] \,|\, \mathcal{B} \leq \mathcal{A}, \mathcal{B}' \leq \mathcal{A}', \mathcal{A}\backslash\mathcal{B} = \mathcal{A}'\backslash\mathcal{B}') \;\geq\; \alpha. \tag{4.1}$$

We refer to the above probability as the semantic equivalence probability.

In other words, if we condition two random ASTs ($\mathcal{A}$ and $\mathcal{A}'$) drawn from the distribution P to (1) contain the subtrees $\mathcal{B}$ and $\mathcal{B}'$ respectively, and (2) agree on global context ($\mathcal{A}\backslash\mathcal{B} = \mathcal{A}'\backslash\mathcal{B}'$), then the probability that the unit test outputs of $\mathcal{A}$ and $\mathcal{A}'$ agree should be high if $\mathcal{B}$ and $\mathcal{B}'$ are semantically equivalent.

Estimating the semantic equivalence probability. Given two subtrees $\mathcal{B}$ and $\mathcal{B}'$, we can now estimate the probability that the two are semantically equivalent. Our method relies on querying the Codewebs index, and we assume queries always return a list of unique ASTs matching a given code phrase, along with a count of the number of submissions matching each AST. First, Equation 4.1 can be written equivalently as:

$$P(F[\mathcal{A}] = F[\mathcal{A}'] \,|\, \mathcal{B} \leq \mathcal{A}, \mathcal{B}' \leq \mathcal{A}', \mathcal{A}\backslash\mathcal{B} = \mathcal{A}'\backslash\mathcal{B}') \tag{4.2}$$

$$\propto \sum_{\substack{\mathcal{A},\mathcal{A}':\\ \mathcal{A}\backslash\mathcal{B}=\mathcal{A}'\backslash\mathcal{B}'}} \mathbb{1}\{F[\mathcal{A}] = F[\mathcal{A}']\} \cdot P(\mathcal{A}|\mathcal{B} \leq \mathcal{A}) \cdot P(\mathcal{A}'|\mathcal{B}' \leq \mathcal{A}). \tag{4.3}$$

```
function EquivalenceProb (Subtrees B, B'):

  L ← QueryIndex(B) ;
  L' ← QueryIndex(B') ;
  Initialize count and denominator to 0;
  Z ← 0;
  foreach AST A_i ∈ L do
      if A_i\B = A'_i\B' for some A'_i ∈ L' then
          w_i ← c_i * c'_i ;
          count ← count + w_i if F[A_i] = F[A'_i] ;
          denominator ← denominator + w ;
  return count/denominator ;
```

Algorithm 1: Pseudocode for estimating the semantic equivalence probability between two subtrees, $\mathcal{B}$ and $\mathcal{B}'$. The function `QueryIndex`($\mathcal{B}$) is assumed to return a list of pairs $(\mathcal{A}_i, c_i)$ such that each $\mathcal{A}_i$ contains $\mathcal{B}$ and c_i is the number of submissions matching $\mathcal{A}_i$. Note that the inner loop can be implemented efficiently using the precomputed hashes of global contexts (described in Section 3).

Estimating Equation 4.3 by plugging in empirical distributions for $P(\mathcal{A}|\mathcal{B} \leq \mathcal{A})$ and $P(\mathcal{A}|\mathcal{B}' \leq \mathcal{A}')$ respectively amounts to summing over pairs of ASTs in the dataset that contain $\mathcal{B}$ and $\mathcal{B}'$, respectively, and match exactly with respect to global context. For intuition, Algorithm 1 shows pseudocode for estimating the semantic equivalence probability for two subtrees using Equation 4.3.

4.2 Semantic equivalence classes and solution space reductions

Having now discussed the problem of verifying semantic equivalence between two given subtrees, we turn to the problem of discovering groups of subtrees that are equivalent and would be useful for canonicalization.

To discover candidate equivalence classes, we use a two stage process: in the first stage an instructor annotates a small set of "seed" subtrees that she believes are semantically meaningful, and in the second stage we algorithmically extract as many semantically equivalent subtrees as possible for each of denoted "seeds."

The main advantage of this semi-autonomous approach is that it results in named equivalence classes (as opposed to anonymous equivalence classes) which can be used to provide non-cryptic feedback. A secondary benefit is that it enables a simple and efficient algorithm to extract the most important equivalences in an intelligent order.

As an example, given the AST $\mathcal{A}$ for the implementation in Figure 1(a), in the first phase an instructor might select the subtree $\mathcal{B}$ corresponding to the expression `(X * theta - y)` and label it as the "residual." In the second phase the algorithm would find other equivalent ways to write the residual, such as `(theta' * X' - y')'`. See Figure 3 for more examples of code snippets that an instructor might select from the linear regression problem.

In phase two, we expand the seed subtrees one at a time to build up an *equivalence class* of subtrees by iterating the following steps:

1. **Finding "feet that fit the shoe".** We perform a first pass to find candidate subtrees that are potentially equivalent to $\mathcal{B}$ by first detaching the subtree $\mathcal{B}$ from its surroundings, leaving the context $\mathcal{C} = \mathcal{A}\backslash\mathcal{B}$. We then query our index for the list of ASTs $\{\mathcal{A}_1, \mathcal{A}_2, \dots\}$ that contain $\mathcal{C}$ (See Figure 2).

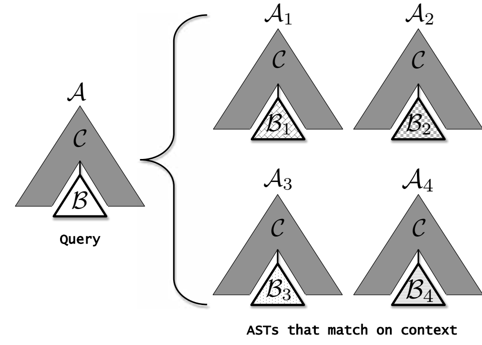


Figure 2: Given a subtree $\mathcal{B}$ of an AST $\mathcal{A}$, we query the Codewebs index for other subtrees that also occur under the same context ($\mathcal{A}\backslash\mathcal{B}$). In this case, $\mathcal{B}_1, \dots, \mathcal{B}_4$ would each be considered as candidates for the equivalence class of subtree $\mathcal{B}$ if the unit test outcomes for their corresponding submissions matched that of the original AST $\mathcal{A}$.

   Under each retrieved AST $\mathcal{A}_i$, the subtree $\mathcal{B}_i$ that is attached to the context $\mathcal{C}$ is then considered to be a candidate for equivalence if $F[\mathcal{A}_i] = F[\mathcal{A}]$ (that is, if the unit test outputs for the containing full AST $\mathcal{A}_i$ agree with those of the original AST $\mathcal{A}$). Intuitively, if another subtree shares the context $\mathcal{C}$ and produces the same output, it suggests that the two can be interchanged without affecting functionality.

   Before we add a new element to our set of equivalent subtrees we use the criterion proposed in Section 4.1 to verify that the new subtree is indeed probabilistically equivalent to another subtree already in the set.

2. **Finding "shoes that fit the feet".** The first step will find all subtrees that have an exact context match with the originally annotated seed subtree. In order to expand our search for further equivalent subtrees, we then search for other contexts that can plausibly be attached to a subtree which is functionally equivalent to the seed.

   For each element $\mathcal{B}'$ in the set of equivalent subtrees found so far, we find all contexts that contain $\mathcal{B}'$ and produce the same output as the program from which we found $\mathcal{B}'$. These contexts are hypothesized to also be attached to subtrees equivalent to the seed, and as such, represent new places to look for more potentially equivalent subtrees.

3. **Repeat Until Convergence.** We repeat steps (1) and (2) and expand both the set of equivalent subtrees as well as the set of the contexts that we believe may be attached to one of the equivalent subtrees until the size of our set stops growing.

Reduction and reindexing. Each equivalence class $\Sigma = \{\mathcal{B}_1, \mathcal{B}_2, \dots\}$ learned from the data yields a set of canonicalization rules. Thus whenever we encounter a subtree $\mathcal{B}$ in an AST which happens to be a member of the equivalence class Σ, we can replace $\mathcal{B}$ by a single member of that equivalence class (say, $\mathcal{B}_1$). The identification of subtrees within an equivalence class leads to a reduction in the complexity of the collection of submitted ASTs since ASTs that were previously distinct can now be transformed into the same tree. Thus after finding an equivalence class, it is easier to expand a subsequent set.

```
theta = theta-alpha*1/m*(X'*(X*theta-y));
```

"residual" "alphaOverM" "hypothesis"

Figure 3: Example code snippets corresponding to AST subtrees tagged by an instructor for the linear regression problem.

{m}

```
m                      rows (X)          rows (y)
size (X, 1)            length (y)        size (y, 1)
length (x (:, 1))      length (X)        size (X) (1)
```

{alphaOverM}

```
alpha / {m}            1 * alpha / {m}       alpha .* 1 / {m}
1 / {m} * alpha        alpha .* (1 / {m})    alpha ./ {m}
alpha * inv ({m})      alpha * pinv ({m})    1 .* alpha ./ {m}
alpha * (1 ./ {m})     alpha * 1 ./ {m}      alpha * (1 / {m})
.01 / {m}              alpha .* (1 ./ {m})   alpha * {m} ^ -1
```

{hypothesis}

```
(X * theta)
(theta' * X')'
[x] * theta
(X * theta (:))
theta(1) + theta (2) * X (:, 2)
        ⋮
sum(X.*repmat(theta',{m},1), 2)
```

{residual}

```
(X * theta - y)
(theta' * X' - y')'
({hypothesis} - y)
({hypothesis}' - y')'
[{hypothesis} - y]
        ⋮
sum({hypothesis} - y, 2)
```

Figure 4: Selections of code snippets algorithmically determined for the linear regression homework. Note that not all subtrees from the equivalence classes are shown.

Example 4. *We highlight several examples of equivalence classes that were automatically discovered from the submissions to the linear regression problem (shown in Figure 4 with names in braces). Equivalence class $\{m\}$, for example, shows that* `length(y)` *and* `size(y,1)` *are equivalent, as discussed in Example 1. Note that equivalence classes can be defined in terms of other equivalence classes.*

It would be difficult to imagine more than a handful of ways to write the expression `alpha/m`*, but our algorithm was able to find 135 distinct AST subtrees in the equivalence class $\{alphaOverM\}$. Note in particular that nearly all of the equivalences are not truly semantically equivalent in the strictest sense of Definition 2, but only in the context of the homework problem. For example,* `alpha/m` *is not interchangeable with* `.01/m` *in general Octave programs, but since* `alpha` *was set to be .01 for the linear regression problem, the equivalence was valid in the sense of Definition 3. Similarly, we inferred an equivalence between the subtrees* `inv(m)` *and* `1./m`*, which would not have been equal if a student had redefined* `m` *to be a matrix instead of a scalar.*

After the determination of each equivalence class, we rebuild the Codewebs index and optionally identify further equivalences. It is often the case that recognizing an equivalence class $\mathcal{E}$ (and reindexing taking $\mathcal{E}$ into account) can help us to discover further equivalence classes. For example, it might be the case that we do not initially have enough observations to conclude with sufficient statistical confidence that `X*theta-y` can be rewritten equivalently as the expression `-(y-(theta'*X')')`. However, by first identifying that `X*theta` can be rewritten as `(theta'*X')'`, we are likely to find more matches in the database when querying for `X*theta-y` and `-(y-(theta'*X')')`, respectively, resulting in a larger sample size from which to draw conclusions. Using the same insight that we leveraged to find equivalence classes, we can also create a class of *attempts* to pair with every equivalence class, where attempts are defined to be subtrees that fit into the same contexts as members of the equivalence class but change the output from correct to incorrect.

5. BUG FINDING FROM SEARCH QUERIES

Since the input to our system includes unit test outputs, determining whether a code submission contains a bug is not a difficult problem. However, determining where the bug lies is considerably more challenging. We discuss a way to use the search engine to localize a bug within code solely by examining its AST. In the next section we will justify its effectiveness by evaluating its ability to determine the presence of a bug in a query code submission *without* having access to the unit test output of the query.

Approach. As our ultimate goal is to provide useful feedback to as many students as possible, we will focus on common, localizable bugs. If we consider the distribution of unit test outputs of ASTs which contain a particular subtree, we would expect that such a distribution corresponding to a buggy subtree would be skewed towards incorrect outputs, while that corresponding to a non-buggy subtree would resemble the overall distribution of unit test outputs. As long as the subtrees are sufficiently small and the bugs sufficiently common, it is possible to have a reliably large sample of these unit test output distributions to use for the purposes of classification.

Notice, however, that as the subtrees become larger, the corresponding sample sizes necessarily decrease. In fact, for any submission not in our training set, the improper subtree consisting of the full AST must has a unit test output distribution with sample size zero. Thus to increase the sample sizes of our distributions, we use distributions corresponding to local contexts instead of subtrees for classification.

Algorithm. Our algorithm consists of an indexing phase and a query phase. In the indexing phase, we iterate over the local contexts of all proper subtrees of all ASTs in our collection. For each unique local context we construct a histogram of the unit tests outputs of the ASTs to which the context belongs. Notice that these histograms can be built via queries to our index described in Section 3. We then classify each local context as being either buggy or not buggy based on the skew and sample size of its unit test output histogram.

In the query phase, we iterate over the local contexts of all proper subtrees of a query AST. For each local context $\mathcal{A}[p[\mathbf{a}]]\backslash\mathcal{A}[\mathbf{a}]$ we mark node $p[\mathbf{a}]$ if the local context is recognized as buggy by our index. We then unmark any node with a marked descendant, and report all remaining marked nodes as the roots of the buggy subtrees.

6. EMPIRICAL FINDINGS

In this section we present empirical studies of the performance of our Codewebs system and use it to study data from Stanford's ML class. Our current implementation uses a parser (adapted from the Octave project [3]) to convert submitted source code to ASTs. The Codewebs indexer can be run on a personal computer with the full index fitting in under 6Gb of main memory for almost all problems. Our running time tests were run on a Lenovo Thinkpad (2.40 GHz) with 8 GB RAM.

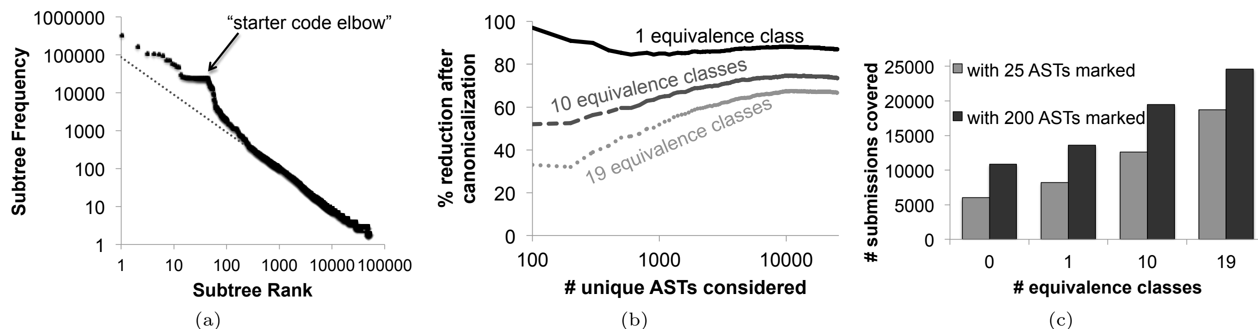


Figure 6: (a) Zipf's Law: subtree frequency plotted against subtree rank (in the frequency table). (b) Fraction of remaining unique ASTs after canonicalizing the k most frequent ASTs with 1, 10 or 19 learned equivalence classes; (c) Number of submissions covered if an instructor were to mark the 25 or 200 most frequent ASTs after canonicalization

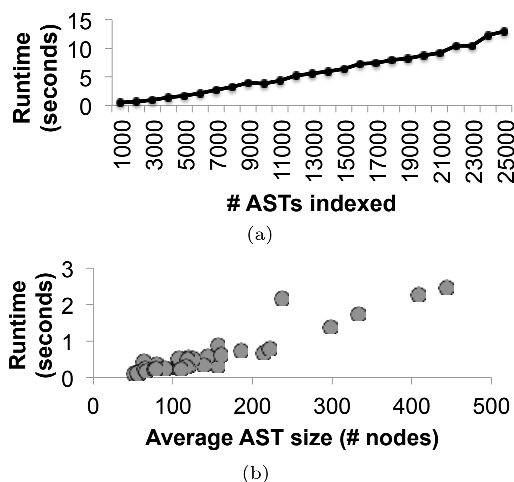


Figure 5: (a) Runtime in seconds for indexing a collection of ASTs (as a function of the number of ASTs) from the "gradient descent (for linear regression)" problem; (b) Runtime in seconds for indexing 1000 ASTs from each of the homework problems for Coursera's ML course plotted against average AST size (# nodes) for each problem

Running time. We first consider the amount of time required for indexing by plotting the running time as a function of the number of ASTs (for the linear regression problem). We index only the uncanonicalized ASTs for this plot, though in general we iterate through multiple stages of indexing as new equivalence classes are discovered. In Figure 5(a), we see that indexing the full set of ASTs requires roughly 16 seconds, excluding the time required for loading the ASTs from disk (which can dominate indexing time, depending on the platform).

We next show how the indexing time scales as a function of the number of nodes in an AST. Figure 5(b) plots the running time required to index 1000 ASTs on forty different programming problems against the average number of nodes per AST for each problem.

We finally note that in all cases, the combined time for all parsing, indexing, and equivalence discovery operations is dominated by the amount of time that students are given to complete an assignment (consisting of 5-7 programming problems on average).

Code phrase statistics. Given the scale of our data, it is natural to ask: *have we seen all of the code phrases that we are likely to see?* For insight, we turn to Zipf's law [26], which characterizes a phenomenon that the frequency of a word in a large scale natural language corpus is inversely proportional to its rank in the frequency table. Thus a few words might be used millions of times, but many words are just used once. Zipf's law has had implications for search engine design, particular for efficient index compression and caching [2]. Among other things, we know that when words are sampled from a Zipfian distribution, the size of the vocabulary grows indefinitely without converging to a fixed maximum size [15].

Asking the similar question of whether code phrases also obey such a law may yield insight into how the Codewebs index is expected to grow as a function of the size of the dataset of submissions. Due to the constrained nature of the data, we might expect significantly less variety in code phrases than in natural language. However, our data tells another story: Figure 6(a), plots the frequency of a subtree against its corresponding rank in the frequency table. This frequency table was constructed from all subtrees taken from 10,000 distinct ASTs submitted to the linear regression problem. The relationship between the log rank and log frequency is evidently nearly linear with an "elbow" at around the 45th most frequent subtree, up to which all earlier subtrees occur more than 10,000 times. We hypothesize that this elbow is due to the subtrees included in the provided starter code that was shared by almost all implementations.

A linear regression shows that $\log(\texttt{rank}) \approx a - b \cdot \log(\texttt{freq})$, where $a = 10.677$ and $b = 1.01$ (which is remarkably close to the slope of 1 postulated by Zipf's law). Thus given a subtree from a new submission, there is a significant probability that it has not yet been observed in the data. On the other hand, the result also suggests that prior work on dealing with large scale traditional indexing may apply to scaling up code phrase indexing as well.

Equivalence classes and reduction. We now evaluate the amount of reduction obtained via canonicalization. We manually tagged 19 code phrases that were likely to vary in the linear regression problem (including those shown in Figure 4) and used the Codewebs engine to thus find 19 equivalence classes. We first examine the reduction factor in the number of distinct ASTs if applying canonicalization to the k most frequently submitted ASTs. Figure 6(b) shows the

result when canonicalizing with just one equivalence class, with 10 equivalence classes, and all 19. We see that using more equivalence classes helps for reduction, and in general, we observe better reduction factors among the more popular ASTs compared to that of the the overall dataset.

We can also examine the number of submissions that an instructor could "cover" by giving feedback only to the k most frequent ASTs (rather than the entire set of ASTs).[1] Figure 6(c) plots the achieved coverage if an instructor were to mark 25 ASTs or 200 ASTs after using canonicalization (again with varying numbers of equivalence classes). Unsurprisingly, the instructor increases coverage simply by marking more ASTs. However, the plots also suggest that canonicalizing has even more of an impact on coverage — we cover nearly 25,000 of the roughly 40,790 submissions to the linear regression problem by marking 200 canonicalized ASTs. Note that we also cover 73% more submissions by marking just 25 canonicalized ASTS than if we were to mark 200 uncanonicalized ASTs.

Bug identification. Evaluating bug localization is challenging since there is currently no available "ground truth" data on the locations of bugs in submitted code. We thus only evaluate our ability to detect the presence of a bug in a submission (instead of requiring localization), for which we do have ground truth through unit tests. Using a $\text{Beta}(0.1, 0.1)$ prior on the probability of an AST having no bugs, we train our detector in leave-one-out fashion, evaluating accuracy of predicting the presence of a bug on the left out example. As a baseline, we compare against a k-nearest neighbor classifier based on tree edit distance between ASTs with $k = 5$ (see our previous work, [10]).

Considering buggy programs as positive examples, we evaluate the precision and recall of both classifiers on the 5000 most frequent ASTs for each homework problem in the class and compute the *F-score* (i.e., the harmonic mean of precision and recall, with higher values corresponding to higher accuracy) in each case. Figure 7(a) compares this F-score performance of our query based bug detection algorithm against that of kNN. As a rough measure of problem complexity, we also visualize the average number of nodes per AST in each problem (after subtracting the number of AST nodes that come from the provided starter code) by varying circle sizes. As can be seen in the plot, bug detection is generally more difficult for both approaches on the more complex assignments, with the neural network training problems being the most difficult. However in many problems, we obtain reasonable accuracy using both approaches with the Codewebs based algorithm outperforming the baseline in most cases.

Figure 7(b) plots the performance of our detection algorithm on the 100 most frequent ASTs, 200 most frequent, and so on for the linear regression problem. As in our reduction experiments, performance is better for the more frequently submitted ASTs, and the drop in accuracy as we get to less frequent ASTs is graceful. Using canonicalization again helps to boost performance in general. On 1000 ASTs, our bug detection precision (with canonicalization) is .78 and recall is .88. Finally, we remark that the bug localization algorithm is fast and capable of handling over 150 requests per second (for the linear regression problem with an average AST size 116).

[1] As we discuss later in this section (the *Feedback case study*), we would more typically mark code phrases rather than full ASTs, which would in general lead to greater coverage of students.

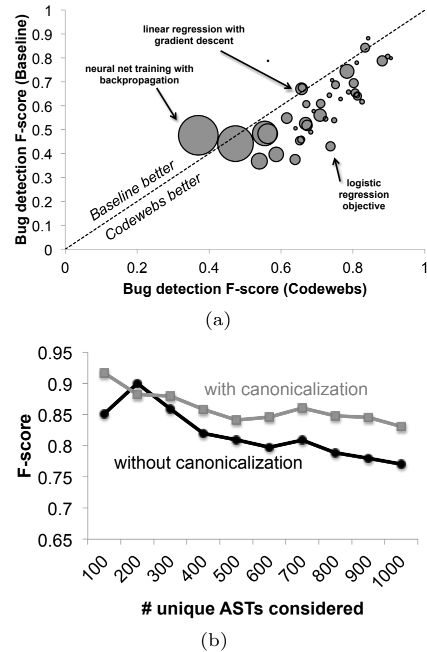


Figure 7: (a) F-score comparison of Codewebs based bug detection algorithm against baseline (5-nearest neighbors) for the 5000 most frequent ASTs for each assigned homework problem. Each circle corresponds to a single homework problem, with circle widths set to be proportional to the average # of nodes per AST for that problem; (b) Codewebs based bug detection F-scores on the k most frequent ASTs, with and without canonicalization on the "linear regression with gradient descent" homework problem.

Feedback case study. Without the help of our system, the easiest way to scalably provide feedback to students in a MOOC is to send canned feedback for the most frequent (incorrect) unit test outcomes, which often reflect common misunderstandings. One weakness of this unit-test based approach, however, is that it can fail when a single conceptual misunderstanding can lead to multiple unit test outcomes, or when it occurs in the presence of other bugs. We illustrate how the Codewebs system can be used to give feedback to students even when the unit-test based approach fails.

In the linear regression problem, many students erroneously included an "extraneous sum" in the gradient expression (e.g., with `sum((X*theta-y)'*X)` instead of `X'*(X*theta-y)`). Upon identifying a single instance of the bug, an instructor can then use the Codewebs system to extract many equivalent ways of writing the erroneous expression (e.g., `sum(((theta'*X')'-y)'*X)`, `sum(transpose(X*theta-y)*X)`, etc.). These expressions are then matched to incoming student submissions, for which we can provide a human generated message explaining the cause of the misunderstanding.

After extracting equivalences and matching to the existing submissions, our algorithm found 1208 submissions which matched the "extraneous sum" bug, outperforming a unit test based feedback strategy which would have covered 1091

submissions. We can also use both strategies together, giving feedback to submissions found by either method to contain the bug, which would lead to a 47% increase in number of submissions covered over just using unit tests.

7. RELATED WORK

Our paper builds upon a body of work on reasoning with code collections — a problem that arises both for programmers and for those learning to program. Code search engines, many of which scale to massive databases have been proposed, for example, for the purposes of code reuse or navigating a complicated API [23, 9]. A number of choices exist in this domain of reasoning with a code collection. How much structure to take into account is one such choice, with some systems reasoning at the level of keywords or tokens [11, 8] to other systems reasoning with the full syntactic structure of the AST [17, 24, 12]. Canonicalization has been a popular approach for reducing the variance of a large collection of code in many such works [1, 25]. In contrast to our paper, this work on code search engines generally has focused on helping programmers to understanding what tools in an existing codebase can help a programmer to accomplish a certain task, which is reflected by features that are commonly supported, such as the ability to query for usage examples for a function, or to find methods that require a certain type as a parameter. Closer to our setting is the work of Rivers et al. [20, 21], who also used ASTs with canonicalization to map out the solution space to introductory programming problems and discover common errors.

However, reasoning about function is critical in many settings, and thus another choice that has received recent attention is whether to incorporate unit test information [11, 14]. Our work efficiently combines what we believe to be the best of the above ideas: reasoning with ASTs and unit tests, as well as combining the two sources of data to automatically discover semantically equivalent code phrases. To our knowledge, this is the first method for automatically finding canonicalization rules for programs.

The idea that student code can be "clustered" into a small category of approaches has also been explored by a number of researchers. Piech et al. [19], for example, cluster trajectories of ASTs over multiple submits by a single student. Glassman et al. [5] visualize the space of student solutions to Matlab programming problems in order to identify popular approaches for solving a problem. A number of recent papers have clustered students based on abstract syntax trees using distances in feature space [7, 6], string edit distance [20, 21] and tree edit distance [10], proposing to give feedback to exemplar submissions. These methods rely almost completely on syntactic analysis and do not explicitly relate form to function as in our work. Furthermore, for assignments with multiple "parts", each of which can be approached in multiple ways, the number of clusters intuitively can grow exponentially in the number of parts, leading to a loss of interpretability and effectiveness of the method. Our work is the first to explicitly address the idea that smaller parts of code can be shared among submissions.

8. DISCUSSION

MOOC platforms now collect massive datasets of student submissions across hundreds of courses, introducing new research problems of how to organize and search the data effectively, but also new opportunities to use the data in ways that were not previously possible. We have developed a system called Codewebs which efficiently indexes all of the code submissions to a MOOC programming assignment and can be useful in a number of applications. Through a novel use of unit test information, our system is also able to determine when code snippets are semantically equivalent in a data driven way.

As the MOOC ecosystem continues to quickly expand, it is crucial for accompanying learning technologies to be applicable to a large variety of courses with minimal effort on the part of the instructor. Codewebs makes no assumptions about the underlying assignments and is designed to be easily applicable to a wide variety of programming languages. Instead of running dynamic analysis on fully instrumented programs, the system relies only on unit test information which is lightweight and thus contributes to portability. To apply Codewebs to a programming problem, the most important requirement is a large dataset of student submissions (which is a given in MOOCs). Beyond data, we only have a handful of requirements: (1) a mechanism for parsing source code into an abstract syntax tree (which is available for most programming languages) (2) a specification of which nodes of an AST correspond to statements, identifiers, and constants, (3) a listing of reserved functions and identifiers that are not to be anonymized, (4) sufficiently informative unit tests for problems, and (5) instructor specified points of variation for determining canonicalization rules. Thus we believe that applying the Codewebs system to submissions in other coding intensive courses should be straightforward.

There are a number of directions ripe for future work. At the moment, our system only handles code that parses correctly. However for beginning programmers, even writing syntactically valid code can be a challenge. Thus the question of how to leverage a large dataset for giving feedback to unparseable submissions remains an important open problem. Our current approach is also limited to indexing submissions of a single function where all implementations receive the same input and thus can be unit tested in a unified way. Thus another open problem is how to deal with long form programs in which students are free to choose how to decompose an implementation into smaller modules.

We believe many of the insights that went into creating the Codewebs system may also apply to general search engines for source code outside of the educational setting. But more interestingly, these ideas may also apply to indexing formal logic proofs [4] and equations [13], or even natural language mathematical proofs, which can be useful either for researchers or educators. Looking even further, it is tantalizing to speculate about the role that effective search engines for student content might play in the MOOC ecosystem beyond STEM (science, technology, engineering and mathematics) courses. Public exhibitions of student work are, for example, a common way to culminate university level art and design courses — but pulling these exhibitions off at the scale of MOOCs requires an effective way to search through tens of thousands of submissions. And while the specific features of designing a search engine for one of these courses will surely be different from the Codewebs setting, many of the same questions will still apply. What are the natural queries for an assignment? How can we exploit the organizational structure of a search engine index to facilitate student feedback? How do we scale to tens of thousands of students?

Acknowledgments

We thank Andrew Ng, Daphne Koller, John Mitchell, and Mehran Sahami for useful conversations. This research was funded by NSF grants CCF 1011228 and DMS 1228304, AFOSR grant FA9550-12-1-0372, a Google Research Award, and the Max Planck Center for Visual Computing and Communications. J. Huang is supported by the CRA CI Fellows grant 104672, and C. Piech by the NSF Graduate Research Fellowship.

9. REFERENCES

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 368–377. IEEE, 1998.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

[3] J. W. Eaton, D. Bateman, and S. Hauberg. *Gnu octave*. Free Software Foundation, 1997.

[4] E. Fast, C. Lee, A. Aiken, M. Bernstein, D. Koller, and E. Smith. Crowd-scale interactive formal reasoning and analytics. In *UIST: ACM Symposium on User Interface Software and Technology,*, 2013.

[5] E. L. Glassman, N. Gulley, and R. C. Miller. Toward facilitating assistance to students attempting engineering design problems. In *Proceedings of the ninth annual international ACM conference on International computing education research*, pages 41–46. ACM, 2013.

[6] S. Gross, B. Mokbel, B. Hammer, and N. Pinkwart. Towards providing feedback to students in absence of formalized domain models. In *Artificial Intelligence in Education*, pages 644–648. Springer, 2013.

[7] S. Gross, X. Zhu, B. Hammer, and N. Pinkwart. Cluster based feedback provision strategies in intelligent tutoring systems. In *Intelligent Tutoring Systems*, pages 699–700. Springer, 2012.

[8] 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.

[9] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22. ACM, 2007.

[10] J. Huang, C. Piech, A. Nguyen, and L. Guibas. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*, page 25, 2013.

[11] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[12] J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an intelligent code search engine. In *AAAI*, 2010.

[13] M. Kohlhase and I. Sucan. A search engine for mathematical formulae. In *Artificial Intelligence and Symbolic Computation*, pages 241–253. Springer, 2006.

[14] O. A. Lazzarini Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 476–482. ACM, 2009.

[15] L. Lü, Z.-K. Zhang, and T. Zhou. Zipf's law leads to heaps' law: Analyzing their relation in finite-size systems. *PloS one*, 5(12):e14139, 2010.

[16] L. Pappano. The Year of the MOOC. New York Times, 2012.

[17] S. Paul and A. Prakash. A framework for source code search using program patterns. *Software Engineering, IEEE Transactions on*, 20(6):463–475, 1994.

[18] C. Piech, J. Huang, Z. Chen, C. Do, A. Ng, and D. Koller. Tuned models of peer assessment in MOOCs. In *Proceedings of The 6th International Conference on Educational Data Mining (EDM 2013)*, 2013.

[19] 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.

[20] K. Rivers and K. R. Koedinger. A canonicalizing model for building programming tutors. In *Intelligent Tutoring Systems*, pages 591–593. Springer, 2012.

[21] K. Rivers and K. R. Koedinger. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, page 50, 2013.

[22] 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.

[23] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In *Proceedings of the 28th international conference on Software engineering*, pages 905–908. ACM, 2006.

[24] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[25] S. Xu and Y. San Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *Software Engineering, IEEE Transactions on*, 29(4):360–384, 2003.

[26] G. K. Zipf. Human behavior and the principle of least effort. 1949.

[27] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):6, 2006.