AutoStyle: Toward Coding Style Feedback At Scale

Joseph Moghadam
UC Berkeley
jmoghadam@berkeley.edu

Rohan Roy Choudhury
UC Berkeley
rrc@berkeley.edu

HeZheng Yin
The Institute for Theoretical Computer Science, Institute for Interdisciplinary Information Sciences, Tsinghua University
yhz11@mails.tsinghua.edu.cn

Armando Fox
UC Berkeley
fox@berkeley.edu

L@S 2015, Mar 14–18, 2015, Vancouver, BC, Canada.
ACM 978-1-4503-3411-2/15/03.
http://dx.doi.org/10.1145/2724660.2728672

Author Keywords
coding style; MOOCs; autograding

ACM Classification Keywords
K.3.2 [Computer and Information Science Education]: Computer science education.

Abstract
While large-scale automatic grading of student programs for correctness is widespread, less effort has focused on automating feedback for good programming *style:* the tasteful use of language features and idioms to produce code that is not only correct, but also concise, elegant, and revealing of design intent. We hypothesize that with a large enough (MOOC-sized) corpus of submissions to a given programming problem, we can observe a range of stylistic mastery from naïve to expert, and many points in between, and that we can exploit this continuum to automatically provide hints to learners for improving their code style based on the key stylistic differences between a given learner's submission and a submission that is stylistically slightly better. We are developing a methodology for analyzing and doing feature engineering on differences between submissions, and for learning from instructor-provided feedback as to which hints are most relevant.

Introduction

Massive Open Online Courses (MOOCs) provide a powerful way to reach thousands of students simultaneously. UC Berkeley's SaaS MOOC "Engineering Software as a Service", based on a campus course for juniors and seniors, teaches fundamentals of software engineering. The MOOC relies heavily on autograding of complete programming assignments. However, good software engineering also requires developing good coding style, and the current autograders only evaluate correctness and completeness. We would like to automatically provide learners with hints for improving their coding style.

By "style" we refer not to mechanical coding conventions (indentation, punctuation, naming of variables, and so on), but to the effective use of programming idioms [14] that make code readable, concise, and elegant by the standards of an experienced practitioner of a given language or framework. For example, consider a simple programming assignment that groups strings in an array so that all strings in a group are anagrams of each other. In a corpus of about 800 submitted solutions from a recent offering of the SaaS MOOC, correct solutions varied in length by an order of magnitude: Figure 1 shows actual submissions from pseudonymous students Alice, Bob, and Charles.

```
def combine_anagrams(words)
  words.group_by { |word| word.chars.sort }.values
end
```

```
def combine_anagrams(words)
  dict = {}
  words.each do |word|
    letters = word.downcase.each_char.sort
    if dict.has_key?(letters) then
      dict[letters] += [word]
    else
      dict[letters] = [word]
    end
  end
  return dict.values
end
```

```
def combine_anagrams(words)
  rtn = Array.new
  words.each do |word|
    p(word)
    wordDowncase = word.downcase
    letters = wordDowncase.split("")
    exist = false
    rtn.each do |rtnAry|
      rl = rtnAry[0].downcase.split("")
      if (rl.length == letters.length) then
        p(rl)
        rl.sort!
        letters.sort!
        match = true
        i = 0
        rl.each do |rli|
          p(((rli + "_") + letters[i]))
          match = false if (rli != letters[i])
          i = (i + 1)
        end
        if (match == true) then
          (rtnAry << word)
          exist = true
        end
      end
    end
    (rtn << [word]) if (not exist)
  end
  return rtn
end
```

Figure 1: A 3-line solution from "Alice," representative of the best possible solution for this assignment; a 12-line solution from "Bob"; and a 30-line solution from "Charles." While longer solutions were more likely to be clumsy and riddled with stylistic problems, stylistic variation was present even among shorter solutions, and excessive terseness is often worse than verbosity, so program length is an insufficient proxy of code style.

Simply showing Charles the solutions from Alice or Bob is not useful, because too many conceptual gaps separate those from his own 30-line solution. Instead, we present a method for automatically using the corpus of submissions to provide Charles with *actionable* style hints—ones that name a specific concept, language construct, or standard-library facility—that, if properly followed, could improve the style of his code incrementally.

Problem Formulation

At the level of an individual function or method, we can quantify code style using well-known software engineering metrics such as ABC complexity (assignments, branches, conditions) [3], absence of redundant code blocks [1], and cyclomatic complexity [8], which measures the number of distinct paths through a piece of code. Existing static analysis tools readily compute these metrics, but do not suggest solutions to the problems they report. Indeed, as the example above suggests, in many cases the "solution" lies not in simple substitution of one token or keyword for another, but of substitution of one idiom or data structure for another.

We hypothesize that with a large enough (MOOC-sized) corpus of submissions to a given programming problem, we can observe a range of stylistic mastery from naïve to expert, and many points in between, and that we can offer advice to a learner by extracting hints from other submissions in the corpus that are "just a little better" than that learner's own submission. Specifically, we use automated techniques to find a "chain" from Charles's solution to Alice's. At each link in the chain, we identify a specific language concept or idiom that is introduced and improves the code's style, while bounding how much the overall code structure changes at each point.

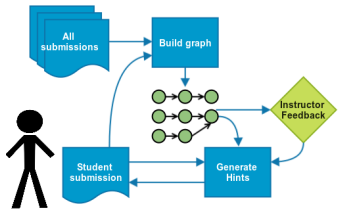

Figure 2: We generate a graph of stylistic improvement across all submissions, which is used to help generate hints for any particular submission. The instructor can provide feedback to train the hint generating system to react to the features of the graph differently.

Our approach requires a measure of structural similarity between two moderate-length pieces of code and one or more measures of stylistic quality. We currently use Zhang-Shasha tree-edit distance [16] as our similarity metric and a variant of ABC (assignment-branch-condition) complexity as a proxy for stylistic quality, but we are experimenting with other choices.

Our approach assumes that all solutions in the corpus are functionally correct—easily enforceable by our existing autograders—and that at least one submission represents the best possible style achievable for the given assignment, as measured by whatever quality metric is used. We call these *stylistically optimal* solutions and recognize that multiple distinct solutions may achieve the same level of code quality.

We first compute the stylistic quality of every submission and the structural similarity between every pair of submissions. We then construct a directed graph of the submissions such that an arc $A \mapsto B$ is present iff B differs structurally from A by no more than a preset threshold, *and* B's quality score improves on A's by at least a preset threshold. (We say "improves on" because lower is better for some metrics whereas higher is better for others.)

Figure 2 illustrates how we envision using this graph in an instructional setting. From any given learner's submission, we find a path through the graph starting at that submission that eventually reaches a stylistically optimal submission; the intuition is that since each successive link in that chain differs from its predecessor by a bounded amount and improves the style of the submission, the chain represents a possible path of "successive refinement" from the learner's submission to a stylistically optimal one. The goal is to identify which of the various structural differences between adjacent nodes is the "best" hint to give the learner to nudge him or her along that path.

Markers

Manual examination revealed that a common pattern underlying needlessly complex code was the learner's lack of awareness of the language's support for a particular data structure, set of operations, or library functions, leading them to code the functionality by hand. Building

on standard Abstract Syntax Tree (AST) analysis libraries, our code looks for *markers* such as the presence of particular calls to the standard library, the presence of local groupings of calls to the standard library, and the presence of certain data structures or operators/library calls that manipulate them. We note which markers are present at each link in the chain (the list of *marker diffs* between adjacent links). We must then decide which marker diffs should be the basis of a hint, which might take the form "Consider using library function *x* here."

Observe that the usefulness of a particular marker-based hint may be specific to a particular programming assignment. The example given earlier can be solved easily with hash tables, but only awkwardly if learners are unfamiliar with hash tables and use linear arrays. For this particular assignment, then, hints based on markers for hashes and their associated operators are likely to be useful.

We say a marker is more *relevant* if a hint based on that marker, properly instantiated, is more likely in the instructor's opinion to call attention to the most important stylistic deficiency in the current submission. We frame the problem of determining the relevance of marker diffs as a classification problem solved with a perceptron [9], where each marker diff is classified as relevant or not. The features the pereptron considers for each marker have to do with the presence of the marker across the chain of submissions associated with the learner's submissions. The reason we construct an entire chain of length n starting at the learner's submission is that one possible indicator of a marker's relevance is its *persistence:* if a marker is absent from submission i in the chain, indicators for its presence in submissions $i+1$, $i+2$, $\ldots$, n are each a distinct feature. This allows the perceptron to learn, for example, that it should prefer hints of concepts that do not appear in the current submission but appear in later submissions in the chain *and persist* after they appear. We initialize the perceptron to already know this, though it can learn more refined behavior for specific markers through the feedback it receives.

Instructor-Facing GUI

We have implemented the analyses described above as well as an instructor-facing GUI for embedding the analyses in an instructional setting. First, an offline batch computation generates the ASTs, quality scores, and pairwise similarities between submissions in the corpus as described above. Then, as the screenshot in Figure 3 shows, the instructor can visualize the chain associated with a given submission and the marker diffs between adjacent submissions in the chain. The instructor can also use sliders to change hyperparameters of the chains, such as adjusting the thresholds for similarity and style improvement: a tighter threshold for structural differences means that instantiating the hints will require the learner to make only minor changes in the code, while a looser threshold implies larger changes in the code.

The instructor provides feedback about which hints are not helpful in the example chains. The tool learns from this feedback by adjusting the perceptron weights, and adjusts the hints. When the instructor is satisfied with the automated hint generation, the tool is used to generate hints for students. The students can then resubmit their assignments; the revised submissions are incorporated into the corpus, and the process is repeated.

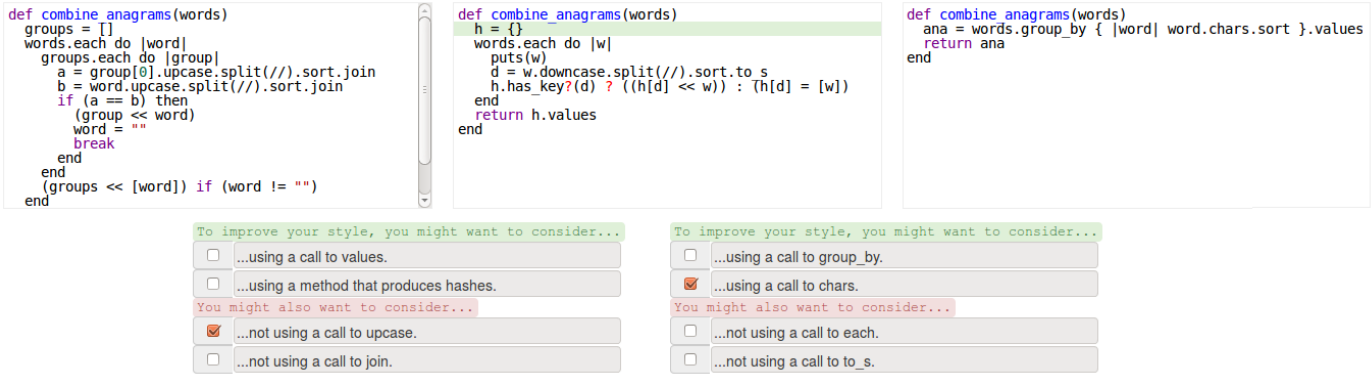

Figure 3: This instructor-facing GUI used to train the perceptron, as described in the text.

Related Work

One of the earliest tools for checking programming style (albeit at a low level of abstraction) is `lint` [6], a static checker that used a set of rules to detect suspicious constructs in C code. The Wikipedia entry *List of tools for static code analysis* lists dozens of `lint`-inspired tools for other languages.

More recently, the intelligent tutoring systems community has focused on generating hints from student solution data by canonicalizing or abstracting aspects of the solution space and using solution-space traversal techniques [11], code synthesis techniques [13], or transformation of the original student submissions [15] to generate the hints. This work generally focuses on helping students turn their incorrect solutions into a correct solution; we focus on proper use of programming idioms, which is both somewhat more subjective and evolves over time. Closer in spirit to our work is Codex [2], which indexes public repositories of Ruby code to extract emergent programming practices, which we can think of as representing "stylistic trends" in a language.

One of the first attempts to visualize variation among submissions to a particular programming assignment from a MOOC cohort was by Huang et al. [5]. Glassman et al. proposed using variation theory to cluster student solutions to an assignment [4], and subsequently produced OverCode [12], a GUI that helps visualize major groupings of variation among student submissions to a code assignment. Codewebs [10] tries to determine "probabilistic semantic equivalence" of two pieces of code by comparing "code phrases," or subgraphs of ASTs. Both of these inherit from much previous work using AST analysis to detect duplicated code or code plagiarism [7].

Status and next steps

We have been developing the tool against one corpus of about 800 submissions to a simple Ruby programming assignment and another corpus of about 200 submissions to a simple Python assignment from an introductory-level

course. The front-end parsers and marker extractors are different for Ruby and Python, but the hint generation and classification logic are the same. We expect to alpha-test the tools with TAs and students in Spring 2015.

Acknowledgments

We thank Andrew Huang for providing anonymized submissions from Berkeley's introductory programming course, and our colleagues Ken Goldberg, Björn Hartmann, and John Canny for feedback on the approach and paper.

References

[1] Baxter, I. D. Clone detection using abstract syntax trees. In *Proceedings of ICSM'98*, IEEE (1998).

[2] Fast, E., Steffee, D., Wang, L., Brandt, J., and Bernstein, M. S. Emergent, crowd-scale programming practice in the IDE. In *CHI 2014* (Toronto, Canada, 2014).

[3] Fitzpatrick, J. Applying the ABC metric to C, C++, and Java. In *More C++ Gems*, R. C. Martin, Ed. Cambridge University Press, New York, NY, USA, 2000, 245–264.

[4] Glassman, E. L., Singh, R., and Miller, R. C. Feature engineering for clustering student solutions. In *Learning At Scale (L@S)* (Atlanta, GA, Mar 2014). Work-in-progress submission.

[5] Huang, J., Piech, C., Nguyen, A., and Guibas, L. Syntactic and functional variability of a million code submissions in a machine learning MOOC. In *MOOCshop Workshop, International Conference on Artificial Intelligence in Education (AIED)* (Memphis, TN, 2013).

[6] Johnson, S. Lint, a C program checker. Tech. Rep. 65, Bell Laboratories, Dec 1977.

[7] Lancaster, T., and Culwin, F. A comparison of source code plagiarism detection engines. *Computer Science Education 14*, 2 (2004), 101–112.

[8] McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering*, 4 (1976), 308–320.

[9] McCulloch, W. S., and Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics 5*, 4 (1943), 115–133.

[10] Nguyen, A., Piech, C., Huang, J., and Guibas, L. Codewebs: Scalable code search for moocs. ACM (2014), 491–502.

[11] Rivers, K., and Koedinger, K. Automating hint generation with solution space path construction. In *12th Intl. Conf. on Intelligent Tutoring Systems* (2014).

[12] Scott, J., Singh, R., Guo, P. J., , and Miller, R. C. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* (2014). Special issue: online learning at scale (to appear).

[13] Singh, R., Gulwani, S., and Solar-Lezama, A. Automated semantic grading of programs. Tech. rep., MIT/Microsoft Research, 2012.

[14] WikipediaContributors. Programming idiom.

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

[16] Zhang, K., and Shasha, D. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing 18*, 6 (1989), 1245–1262.