Scale-driven Automatic Hint Generation for Coding Style

Rohan Roy Choudhury, Hezheng Yin, and Armando Fox

University of California, Berkeley
{rrc,hezheng.yin,fox}@berkeley.edu

Abstract. While the use of autograders for code 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 present a system that can provide real-time actionable code style feedback to students in large introductory computer science classes. We demonstrate that in a randomized controlled trial, 70% of students using our system achieved the best style solution to a coding problem in less than an hour, while only 13% of students in the control group achieved the same. Students using our system also showed a statistically-significant greater improvement in code style than students in the control group.

Keywords: coding style, autograding, automatic hint generation, MOOCs

1 Motivation and Overview

Rapid feedback is integral to mastery learning. Prior work has shown that students learn best through the process of repeatedly submitting, receiving immediate actionable feedback and resubmitting [1, 9, 13]. Automatic graders (autograders) provide this capability and are thus used extensively in programming courses, especially Massive Open Online Courses (MOOCs). However, while the use and development of autograders for code correctness is widespread, less effort has focused on automating feedback for good programming style [17].

Software with poor code quality has been shown to require significantly higher maintenance, a sobering fact considering that maintenance dominates software cost [5]; good coding style therefore has significant implications for the software industry. By providing students with rapid and actionable style feedback, intelligent tutoring systems can help future software developers develop good coding style habits early.

Most existing code style tools check code against a fixed set of style rules that do not depend on the specific code being analyzed. Checkers such as `lint(1)` and `pylint` and existing autograders such as `rag` [6] are unable to account for subtleties such as whether using a different data structure, language construct or library call might be stylistically better, and therefore cannot provide actionable feedback on how to improve style [6, 11]. As a result, providing actionable style feedback usually requires instructors to manually read student code, which can be resource-prohibitive in large courses. Our university's rigorous introductory computer science course relies on over 40 teaching assistants to manually grade over a thousand code submissions per assignment. Given scarce TA resources, style is lightly graded on a coarse-grained scale based on a "style guide" given to students. Automating style grading would save significant instructor time and could provide more tailored feedback to support mastery learning.

Our approach to providing such guidance automatically is to (1) identify similarities among student code submissions for a short assignment (a few lines to tens of lines of code), (2) analyze these similarities using clustering techniques and Abstract Syntax Tree (AST) comparison, and (3) use them to deliver a combination of instructor-authored guidance and auto-generated syntactic hints, such that the guidance provided on a given submission is based on properties of another student's structurally similar but stylistically superior submission.

Specifically, we make the following contributions:

1. Two techniques for analyzing similarities in student code for short assignments: one based on unsupervised classification and the other based on differencing of the ASTs of student submissions.
2. A workflow based on the above techniques that enables instructors to efficiently provide style feedback for a large body of submissions to the same assignment, with effort proportional to the number of distinct approaches to solving the problem, not the number of students.
3. An unsupervised, automated, student-facing workflow that provides students with a combination of instructor-authored guidance and automatically-generated guidance based on similar submissions by other students.
4. A randomized controlled trial experiment demonstrating the efficacy of our system. Students in the treatment group showed a statistically-significantly greater improvement in style than students in the control group.

2 Related Work

Most work on hint generation has focused on code correctness. Lazar and Bratko [14] construct hints for Prolog programs in a generative manner based on specific editing operations that transform the program code. Rivers and Koedinger [19] propose a method for automatic code correctness feedback by using AST differencing to identify a student's state in a solution space and showing the student another student's slightly-better program as feedback, developing various techniques to reduce the vast solution space and make the hint-generation problem tractable. In contrast, we assume students start with a correct but possibly ugly solution, which they may have produced on their own or with the help of such a system and/or verified against a test-based autograder [6].

Whereas early work on providing automated feedback was based on (often manually-constructed) "bug libraries," as large corpora of code have become available (due the increasing class sizes and the availability of cloud services such as GitHub), guidance systems have begun generating feedback by comparing student code to an existing corpus. Codex [2] discovers common language idioms (integral to good style) and detects patterns in the student's code that might benefit from applying them. Codewebs [18] tries to identify semantically-equivalent code blocks in different students' submissions, to which the same instructor feedback can be applied. Both approaches use abstract syntax tree (AST) differencing to compare code exemplars. We use similar techniques to identify correct student submissions that are similar but have salient stylistic differences, and use these submissions to generate style feedback.

We also draw upon recent work on using machine learning techniques to increase instructor leverage. Huang et al. [10] found that clustering ASTs of student submissions produces clusters that embody similar strategies to solving the problem and could potentially receive the same feedback. Glassman et al. [8] hierarchically cluster student submissions, based first on student strategy and then on implementation. They identify the features required for effective clustering. We draw upon their work to cluster existing student submissions to allow instructors to provide predetermined style feedback for students solving the problem using a particular strategy.

3 Approach

We and others have observed that given a large enough corpus of submissions to a given programming problem, there exists a range of stylistic mastery, from naïve to expert [17]. Figure 1 shows three correct submissions from students with pseudonyms Alice, Bob, and Charlie, who provide three correct solutions to the same simple problem: given a list of words, return a list of groups such that all words in each group are anagrams of each other. As the figure shows, correct solutions vary in length (and therefore complexity) by nearly a factor of ten. While we could simply show Alice's solution to Charlie, many conceptual gaps separate her concise solution from his 30-line solution. In contrast, guiding students to incrementally improve and discover the best solution has been shown to be more conducive to mastery learning by reducing cognitive load, especially for struggling students [20]. Thus, we seek a sequence of hints that will guide Charlie to incrementally transform his solution to one like Alice's.

In order to provide style-improvement feedback based on differences between student submissions, we need a way to measure both style goodness and differences. The software engineering literature suggests a variety of metrics of stylistic quality [12]. We have found empirically that the ABC score, which tallies a weighted count of assignments, branches, and conditional statements in a block of code [3], is a good proxy for stylistic quality when used on short (a few lines to a few tens of lines) code fragments. It relies on static analysis only, and is easy to implement and fast to compute. In general, a lower score is better, but it is an ordinal metric, i.e. cutting the ABC score by half does not necessarily imply that the code has doubled in stylistic quality. That said, the choice of algorithm used to compute the quality score is an input to our workflow, and any metric that obeys the triangle inequality can be used.

```
def combine_anagrams(words) #Alice
  words.group_by{|w| w.chars.downcase.sort}.values
end
```

```
def combine_anagrams(words) #Bob
  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) #Charlie
  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
```

Fig. 1: A 3-line correct solution by Alice, 12-line correct solution by Bob, and 30 line correct solution by Charlie to the same problem, illustrating the range of stylistic mastery commonly found in the type of assignments used in introductory classes.

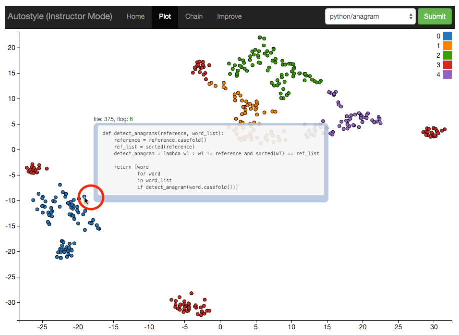
The edit distance between the abstract syntax trees (ASTs) is a common measure of similarity between two code fragments [22]. To emphasize the importance of higher-level structure (the "problem solving strategy"), we use the *normalized* tree edit distance (n-TED) of the AST, which weights nodes closer to the root of the AST more heavily, thus preventing minor syntactic differences at the leaves from affecting the similarity score of programs that are structurally similar, but differ in low-level details [21].

4 Instructor and Student Workflow

Our workflow starts with a corpus of existing submissions to a programming problem, which may include an instructor-authored canonical solution. This corpus may consist of submissions from a previous offering of the course, or it can be bootstrapped using submissions from a subset of the students in a large-enrollment course. We perform an offline computation to generate the AST and quality score for every submission, and the pairwise similarity between all pairs of submissions. The submission(s) with the best style score(s) are judged to be the best possible style exemplars for this problem. The result of this step is an undirected weighted complete graph in which each student submission is a vertex and the tree edit distance between submissions are the weights on the edges.

We then cluster the student submissions to aggregate groups of submissions that use the same problem-solving strategy. We observed that stylistically-better solutions tend to be densely clustered, whereas stylistically weak solutions tend to form sparse clusters (informally, there are many more varied distinct ways to be stylistically "wrong" but only a few ways to be stylistically "right" for a short assignment). We therefore use the OPTICS density-based clustering algorithm [21].

The instructor then annotates each cluster with three items. The first is a label: `good`, `average`, or `weak`. A `good` cluster has solutions close to or identical to the best solution. `Average` clusters contain solutions that solve the problem using a mundane approach and can thus still improve on both approach and language idioms. `Weak` clusters contain solutions that generally exhibit lack of knowledge of one or more important language concepts or constructs that are essential to solving the problem with excellent style. There is clearly instructor subjectivity in applying these labels; to aid the instructor, we display an interactive 2D visualization, as Fig. 2 shows.



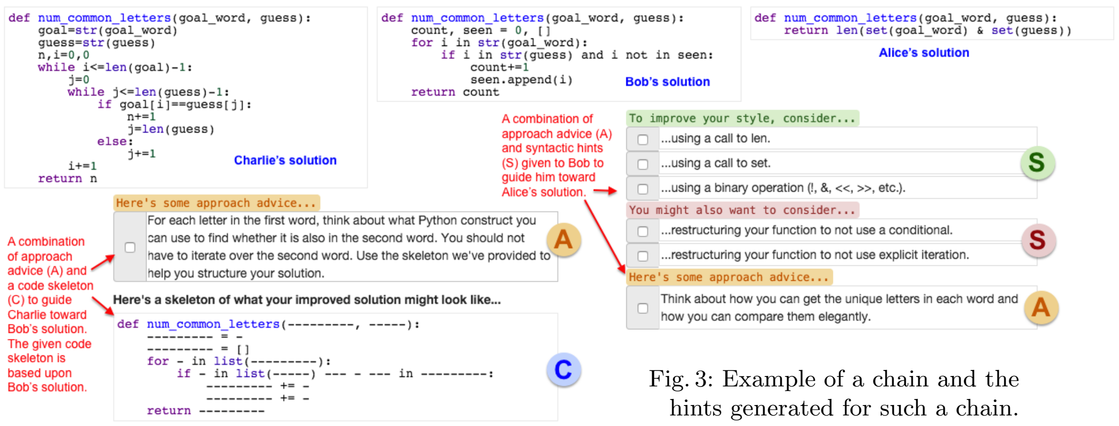
```
def detect_anagrams(reference, word_list):
    reference = reference.casefold()
    ref_list = sorted(reference)
    detect_anagram = lambda w1 : w1 != reference and sorted(w1) == ref_list

    return [word
            for word
            in word_list
            if detect_anagram(word.casefold())]
```

Fig. 2: t-SNE [15] 2D visualization of clustering 425 submissions. Each dot represents a submission, colors represent clusters, and hovering over a dot shows the actual code associated with that submission.

The second item is an *approach hint* for the cluster. Approach hints aim to correct a misunderstanding or lack of awareness of the best way to approach the problem; they illustrate the high-level reasoning of how to approach the problem from a new direction while still leaving the work of developing and implementing a more elegant solution to student. That is, this is the hint that the instructor would give a student whose submission was similar to the cluster members.

The third item is an *exemplar* the instructor chooses from another cluster that she believes to represent a better approach. In keeping with our philosophy of incremental improvement, we ask the instructor not to simply select an exemplar from the "best" cluster as part of the approach hints.

In addition to the instructor-authored annotations on each cluster, our system automatically produces two other types of guidance. *Code Skeletons* are redacted versions of other students' solutions that demonstrate the key control flows and structure of a possible solution, while obfuscating variable names and function call names. *Syntactic hints* guide the student to add (remove) specific structures (loops, conditionals, special language constructs, calls to common built-in or library functions) in order to improve style, based on the presence (absence) of those features in submissions with better style. Syntactic hints are derived by *chain-building* [17], a process that traverses the complete graph generated in the preparation step to find a path from a given submission to one of the "best possible" submissions. The path is subject to the constraints that for each edge $A \rightarrow B$, the n-TED structural difference between A and B does not exceed a set threshold, and B's style score is better than A's by a set threshold. The path is analyzed to determine the most important syntactic hints corresponding to structural features present (absent) in later links in the chain. The feature vectors used in this analysis check for specific language features such as built-in functions, language idioms, and basic control flow constructs in each language; we have constructed feature vectors for Ruby, Java, and Python.



```
def num_common_letters(goal_word, guess):
    goal=str(goal_word)
    guess=str(guess)
    n,i=0,0
    while i<=len(goal)-1:
        j=0
        while j<=len(guess)-1:
            if goal[i]==guess[j]:
                n+=1
                j=len(guess)
            else:
                j+=1
        i+=1
    return n
```

```
def num_common_letters(goal_word, guess):
    count, seen = 0, []
    for i in str(goal_word):
        if i in str(guess) and i not in seen:
            count+=1
            seen.append(i)
    return count
```

```
def num_common_letters(goal_word, guess):
    return len(set(goal_word) & set(guess))
```

Here's some approach advice...

- [ ] For each letter in the first word, think about what Python construct you can use to find whether it is also in the second word. You should not have to iterate over the second word. Use the skeleton we've provided to help you structure your solution.

Here's a skeleton of what your improved solution might look like...

```
def num_common_letters(---------, -----):
    --------- = -
    --------- = []
    for - in list(---------):
        if - in list(-----) --- - --- in ---------:
            --------- += -
            --------- += -
    return ---------
```

To improve your style, consider...

- [ ] ...using a call to len.
- [ ] ...using a call to set.
- [ ] ...using a binary operation (!, &, <<, >>, etc.).

You might also want to consider...

- [ ] ...restructuring your function to not use a conditional.
- [ ] ...restructuring your function to not use explicit iteration.

Here's some approach advice...

- [ ] Think about how you can get the unique letters in each word and how you can compare them elegantly.

Charlie's solution
Bob's solution
Alice's solution
A combination of approach advice (A) and syntactic hints (S) given to Bob to guide him toward Alice's solution.
A combination of approach advice (A) and a code skeleton (C) to guide Charlie toward Bob's solution. The given code skeleton is based upon Bob's solution.
A
C
S
S
A

Fig. 3: Example of a chain and the hints generated for such a chain.

5 Experiment Design and Setup

We performed an intervention experiment using $n = 80$ compensated student participants and compensated teaching assistant participants to evaluate the efficacy of our system under realistic conditions. [1] All recruited participants were associated with our university's large-enrollment introductory computer science course, which introduces a range of programming concepts using the Python language. Participants were recruited by advertising in the course discussion forum and were paid US$15 for one hour of their time.

The primary hypothesis is as follows: Compared with students who are given only a set of "good style" guidelines, *students receiving hints via our automated workflow will improve their code quality more in a given period of time.*

We had a corpus of 265 student submissions of this assignment from a previous offering of the course. Prior to working with the study participants, we ran our clustering algorithm on this corpus and labeled each generated cluster as `good`, `average`, or `weak`; we annotated `average` and `weak` clusters with *approach hints*, and picked *exemplars* for the `weak` clusters. To help validate that the clusters do indeed capture common approaches, we recruited two TAs from the same course and asked each to write down in their own words a description of the overall approach represented by each cluster's members, and two additional TAs to judge whether the descriptions provided by the first two TAs were similar on a five-point scale. We report a square weighted Cohen's kappa of 0.71 and an average similarity rating of 3.85 (σ=0.91). These statistics indicate that different instructors are able to recognize the approaches captured by the clusters.

The recruited students were randomly placed into either the treatment group (50 students) or control group (30 students). Both groups were given the same Python programming assignment, based on a previous offering of the course but absent from the current offering. All participants were provided with the "style guide" authored by the course staff and were allowed access to the Internet to look up documentation. All participants were shown the same problem and in-

[1] IRB Protocol number: 2015-10-8003

structed to submit a solution; participants were allowed as much time as they wanted (within the one-hour time limit of the experiment) to do so. Upon submission, participant solutions were automatically evaluated against a set of test cases for correctness. Upon submitting a correct solution, the participant was immediately shown the computed "style score" for their solution as well as the best possible style score for this problem (2.41 based on the corpus of previous submissions—recall that lower ABC scores are better), and asked to revise their submission to work towards the best score. The control group was given only the style guide (reflecting current practice in the course), whereas the treatment group received specific automatically-generated feedback from our system.

In particular, each submission from a treatment-group student was first analyzed using k-nearest neighbors to determine which cluster it would belong to. If it belonged to a `good` cluster, the participant was shown only a syntactic hint based on building a chain from his submission to the best submission. If it belonged to an `average` cluster, the participant was shown the instructor's approach hint for that cluster, *and* a syntactic hint. If it belonged to a `weak` cluster, the participant was shown the instructor's approach hint for that cluster, *and* the `code skeleton` of the instructor-chosen exemplar for that cluster. Code skeletons are automatically constructed using a regular expression that redacts variables and function call names while retaining control flow structures.

All participants were asked to repeatedly revise their solution based on feedback until they achieved the best possible quality score or exceeded one hour.

6 Results

We collected every correct submission made during the experiment for both groups. Figure 4 shows each student's submission history and the type of feedback they received. There was no significant difference in the style score of the initial submission between the two groups ($p = 0.21$, Pearson's χ^2 test). However, students in the treatment group ended with significantly better style scores ($p = 0.007$, Kruskal-Wallis H test), indicated in the graph by the treatment group vertical lines ending much lower than the control group ones (lower style scores are better with the ABC metric we used).

Figures 4 and 5 show that the percentage of students that achieved the best style solution (style score of 2.41) is considerably greater in the treatment group than in the control group. Moreover, as shown in Fig. 5, students in the treatment group improved significantly more than those in the control group over the one hour experiment period. They also showed significantly more improvement per submission attempt than control group.

To evaluate the effectiveness of the different types of guidance, we asked students to rate the helpfulness of different types of hints on a scale of 1 (not at all helpful) to 4 (very helpful) immediately after completing the study. We find that when students were given different types of hints, neither type of hint was perceived to be significantly more helpful than the others. Specifically, students reported a mean perceived helpfulness of 3.13 ± 0.79 for syntactic hints (S), 2.77 ± 0.89 for approach hints (A), and 2.85 ± 0.82 for code skeletons (C). We

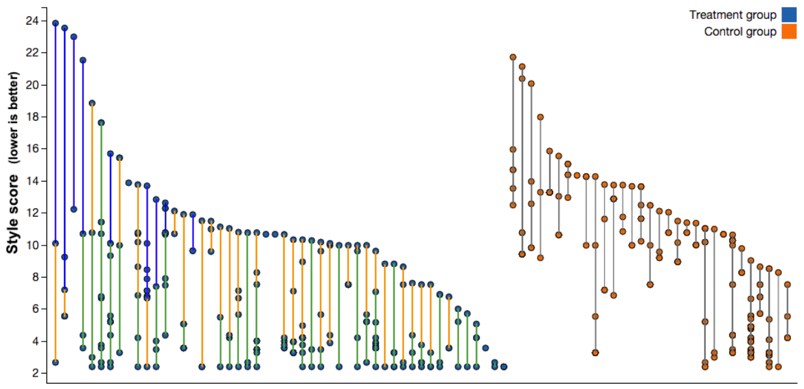


Fig. 4: Each vertical line represents a student and each dot along the line is a submission. The color of line segments between dots for the treatment group codifies the combination of hints the student received—blue: `approach` + `code skeleton`, yellow: `approach` + `syntactic`, green: `syntactic` only.

Metric	Treatment	Control	Statistically significant?
% of students achieving best solution	**70%**	13%	Yes ($p < 0.001$)†
Mean improvement in style score	$\mathbf{7.1 \pm 4.9}$	4.1 ± 3.1	Yes ($p = 0.007$)‡
Mean improvement per attempt	$\mathbf{1.8 \pm 3.12}$	0.62 ± 1.9	Yes ($p < 0.001$)‡

Fig. 5: Key results. †Fisher's exact test ‡Kruskal-Wallis H test

also studied the ratings distribution for the subset of students who received some combination of hints ($A + S$ or $A + C$); at a 5% significance level (t-test), we found no evidence of significant difference between the perceived helpfulness of different types of hints in either group ($p = 0.092$ for $A+S$, $p = 0.760$ for $A+C$).

7 Discussion, Limitations, Assumptions

While we are encouraged by the positive results, we note some caveats and assumptions. First, our chosen metric of style (ABC score) favors a particular definition of style consistent with our own opinions as instructors; different metrics may better suit the needs of other pedagogy. Second, we rely on the instructor to write a good approach hint for a cluster. Third, we assume that the best style solution is represented somewhere in the initial corpus, though this is easily ensured by including the instructor's reference solution. Fourth, although we have tested the clustering and chain-building on other languages and assignments with good results, the current experiments were conducted on a single assignment in one language. Finally, while student feedback on the types of hints suggests that no hint type's usefulness dominates the others, we plan to try to isolate the effects of each in future experiments.

A clear limitation of the current system is its ability to examine only a single function at a time. A standard style guideline is to improve a function by refactoring it to use "helper" functions, but our system cannot currently handle such assignments. We would need to enhance our n-TED similarity metric to account for such submissions.

Our system deliberately provides guidance consistent with two observations about how professional programmers learn. The first is the importance of *concrete rather than abstract advice* for improving coding style. The "style guide" provided to students in the course we worked with can be seen as a microcosm of the well-developed paradigms in software engineering for improving code readability and maintainability, including refactoring and applying design patterns. Yet the canonical reference books on those topics [7, 4] feature an abundance of concrete examples to illustrate the abstract points. We speculate that like the professional programmers who are the target audience of such books, students learn better when a hint or technique is situated in a concrete example, as our hints and code skeletons try to do, rather than stated as an abstract principle.

Second, programming requires *active independent learning.* Following good design principles requires knowledge of language features or library functions of which students may be unaware. Both syntactic auto-generated hints and instructor-authored approach hints can point students in the right direction by suggesting, for example, "Consider using a call to `set()`". Even if a code skeleton is provided with the hint, the skeleton is sufficiently redacted that the student cannot simply copy and paste the code without modification. To improve their code, the student has no choice but to go off and learn about the language feature or library function suggested by the hint or code skeleton, possibly seeking the help of peers or instructors in doing so.

Our system allows instructors and students to enjoy these benefits with a level of instructor effort proportional to the number of clusters, not the number of students. Our system currently focuses on giving feedback for one function or method at a time; since good functions should be short [16], there are only a finite number of strategies that might be used for a function, so we expect the number of clusters to grow very slowly with the number of students. Figure 6 shows that this is indeed the case for seven such assignments we studied.

Class size (number of students)	265	425	448	686	951	986	1607
Number of Clusters	8	3	5	5	3	6	4

Fig. 6: Class size vs. number of clusters for seven comparable assignments.

8 Future Work

We plan to field-test this system in one or more large-enrollment campus courses as well as free Massive Open Online Courses (MOOCs) that teach programming skills. A key question is whether we can observe transfer of improved code style skills after students interact with our system; MOOCs would be an excellent testbed for a randomized controlled experiment to measure transfer.

We have not focused on the relatively well-explored area of generating hints for program correctness, in part because we have observed as instructors that students will first work toward a correct program "by any means necessary" (including with the support of automated hints from an intelligent tutoring system), and only later think about refactoring and improving its style (if they think about these things at all). Indeed, this process is reflected in the "red–green–refactor" cycle [5] espoused by the Test-First Development approach within the Agile methodology: programmers are advised to start with nonworking code that fails a correctness test (red), debug it until it passes the correctness test (green), then refactor the code and design to improve readability and maintainability.

References

1. Ericsson, K., Krampe, R., Tesch-Römer, C.: The role of deliberate practice in the acquisition of expert performance. Psychological review 100(3), 363–406 (1993)
2. Fast, E., Steffee, D., Wang, L., Brandt, J., Bernstein, M.: Emergent, crowd-scale programming practice in the IDE. In: SIGCHI Conference on Human Factors in Computing Systems. Toronto, Canada (2014)
3. Fitzpatrick, J.: Applying the ABC metric to C, C++, and Java. In: More C++ Gems, pp. 245–264. Cambridge University Press, New York, NY (2000)
4. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
5. Fox, A., Patterson, D.: Engineering Software as a Service. Strawberry Canyon LLC, San Francisco, CA (2014)
6. Fox, A., Patterson, D. Joseph, S., McCulloch, P.: MAGIC: Massive automated grading in the cloud. In: CHANGEE (Facing the challenges of assessing 21st century skills in the newly emerging educational ecosystem) workshop at EC-TEL 2015. Toledo, Spain (2015)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
8. Glassman, E., Singh, R., Miller, R.: Feature engineering for clustering student solutions. In: 1st ACM Conference on Learning at Scale. Atlanta, GA (2014)
9. Guskey, T.R.: Closing achievement gaps: revisiting Benjamin S. Blooms "Learning for Mastery". Journal of Advanced Academics 19(1), 8–31 (2007)
10. Huang, J., Piech, C., Nguyen, A., Guibas, L.: Syntactic and functional variability of a million code submissions in a machine learning MOOC. In: International Conference on Artificial Intelligence in Education (AIED). Memphis, TN (2013)
11. Johnson, S.: Lint, a C program checker. Tech. Rep. 65, Bell Labs (1977)
12. Kan, S.H.: Metrics and Models in Software Quality Engineering. Addison-Wesley, Boston, MA, 2nd edn. (2002)
13. Kulkarni, C.E., Bernstein, M.S., Klemmer, S.R.: PeerStudio: Rapid peer feedback emphasizes revision and improves performance. In: 2nd ACM Conference on Learning at Scale. Vancouver, Canada (2015)
14. Lazar, T., Bratko, I.: Data-driven program synthesis for hint generation in programming tutors. In: Intelligent Tutoring Systems. pp. 306–311. Springer (2014)
15. Van der Maaten, L., Hinton, G.: Visualizing data using t-SNE. Journal of Machine Learning Research 9(2579-2605), 85 (2008)
16. Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall (2008)
17. Moghadam, J., Roy Choudhury, R., Yin, H., Fox, A.: AutoStyle: Toward coding style feedback at scale. In: 2nd ACM Conference on Learning at Scale. Vancouver, Canada (2015)
18. Nguyen, A., Piech, C., Huang, J., Guibas, L.: Codewebs: Scalable code search for MOOCs. In: 23rd International Conference on world wide web. Seoul, Korea (2014)
19. Rivers, K., Koedinger, K.R.: Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. International Journal of Artificial Intelligence in Education pp. 1–28 (2015)
20. Shute, V.J.: Focus on formative feedback. Review of educational research 78(1), 153–189 (2008)
21. Yin, H., Moghadam, J., Fox, A.: Clustering student programming assignments to multiply instructor leverage. In: 2nd ACM Conference on Learning at Scale. Vancouver, Canada (2015)
22. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM journal on computing 18(6), 1245–1262 (1989)