AutoStyle: Scale-driven Hint Generation for Coding Style

Rohan Roy Choudhury
Hezheng Yin
Joseph Moghadam
Antares Chen
Armando Fox

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-40
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-40.html

May 5, 2016

Acknowledgement

I would like to thank my advisor, Armando Fox, for all his support, guidance, and encouragement over the past few years. I would also like to thank Antares Chen, Joseph Moghadam, and Hezheng Yin for their invaluable contributions to this project. This work was supported in part by a grant from Google Inc.

AutoStyle: Scale-driven Hint Generation for Coding Style

by Rohan Roy Choudhury

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Armando Fox
Research Advisor

(Date)

* * * * * * *

Professor John DeNero
Second Reader

(Date)

AutoStyle: Scale-driven Hint Generation for Coding Style

Rohan Roy Choudhury
UC Berkeley
rrc@berkeley.edu

Hezheng Yin
UC Berkeley
hezheng.yin@berkeley.edu

Joseph Moghadam
UC Berkeley
jmoghadam@berkeley.edu

Antares Chen
UC Berkeley
antaresc@berkeley.edu

Armando Fox
UC Berkeley
fox@cs.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. We also present experiments to demonstrate the efficacy and relevance of each of the different types of hints generated by our system.

Author Keywords

coding style; autograding; automatic hint generation; intelligent tutoring systems; MOOCs

ACM Classification Keywords

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

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.

Parts of this work have been published in the proceedings of L@S 2015, CSCW 2016, and ITS 2016.

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.

We believe these contributions constitute a novel approach to autograding.

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.

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 [21]. 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.

The edit distance between the abstract syntax trees (ASTs) is a common measure of similarity between two code fragments [24]. 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 [23].

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

```ruby
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
```

```ruby
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
```

Figure 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.

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 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. The instructor then annotates each cluster with three items.

The first item 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.

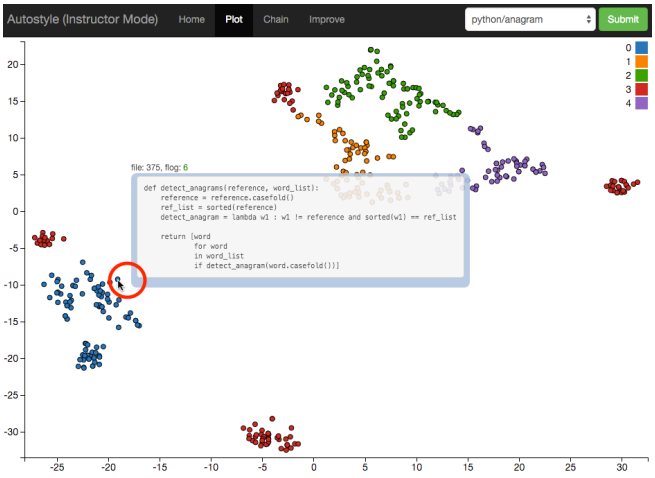


Figure 2: t-SNE [22] 2D visualization of clustering 425 submissions. Each dot represents a submission, colors represent clusters, and hovering over a dot shows the actual code associated with that submission.

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 they believe 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)

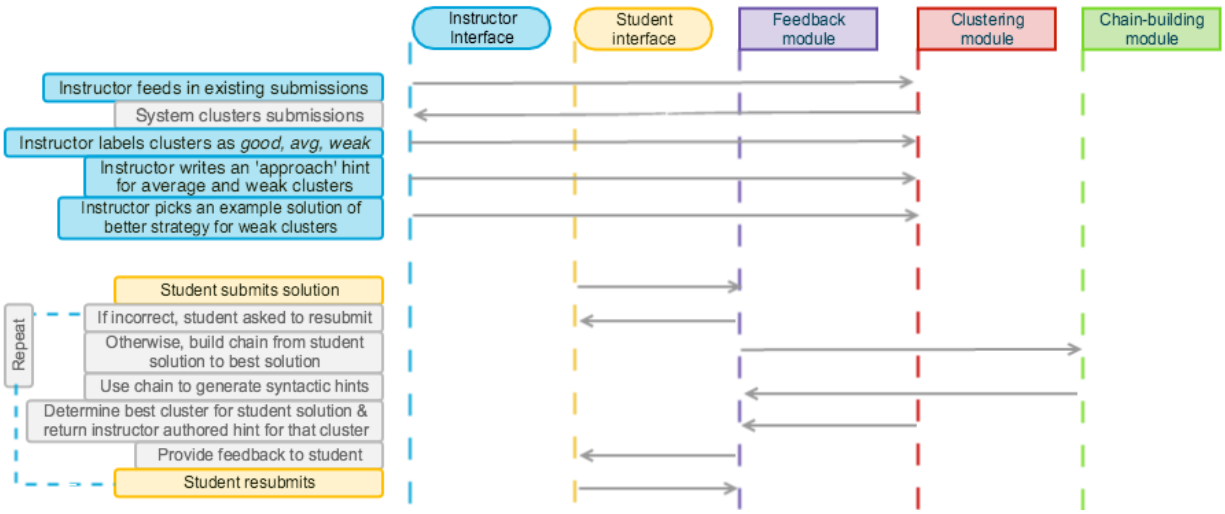


Figure 3: System workflow summary.

in order to improve style, based on the presence (absence) of those features in submissions with better style.

Students use these hints to improve their solutions and resubmit. Upon submitting, they are instantaneously given style feedback. The iterative process of using the feedback and resubmitting continues until the student meets some stopping criteria (style threshold, time limit) determined by the instructor.

CLUSTERING METHODOLOGY

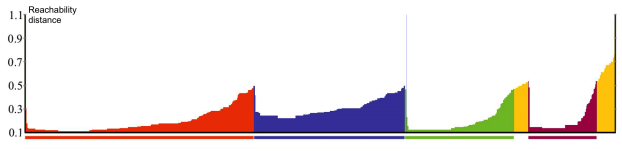
Choice of Algorithm

The goal of our clustering module is to generate clusters such that a human instructor can easily "write down a label" for each cluster that describes the strategy used by submissions in that cluster after viewing only a small subset of submissions for each cluster. We tested several clustering algorithms (k-Means, weighted k-Means, spectral clustering, OPTICS, DBSCAN) on a number of Ruby and Python assignment datasets from computer science classes at UC Berkeley. We measured clustering performance using the silhouette metric [20], a commonly used measure of cluster quality. We observed that density-based clustering algorithms yield the best clusters [23]. This is intuitive—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). This observation informed our decision to use the OPTICS density-based algorithm as our clustering algorithm.

Similarity Metric

We use the normalized tree edit distance (n-TED) of the ASTs as our similarity metric for clustering. Cognizant of the top-down structure characteristic of programming, n-TED emphasizes the importance of high-level program structure. It assigns heavier weights to nodes higher (closer to the root) in the AST. By doing so, it prevents minor differences in syntax (i.e. code at the leaves of the tree) from affecting the similarity score of code fragments that are structurally similar but differ in low-level implementation. We use a dynamic-programming algorithm by Zhang and Shasha [24] to compute n-TED.

We found that n-TED outperforms standard TED when used as a similarity metric for clustering. Figure 5 includes reachability plots for an OPTICS clustering of student solutions to a programming assignment typical of those found in software engineering courses. The reachability plots in Figure 5 show that the clusters formed using n-TED are much denser, indicated by the deeper "valley-like" structures in (a). The "valleys" result from points belonging strongly to one cluster (thus having a low reachability distance to their nearest neighbor). TED also has many more outlier points – points that do not fit into any cluster (indicated in yellow). Figure 5 shows that n-TED led to more stable clusters and fewer outliers than TED.



(a) n-TED

(b) TED

Figure 5: Reachability plots for TED and n-TED for a Ruby programming assignment with 425 submissions clustered using OPTICS. Colors represent clusters. Yellow represents outliers.

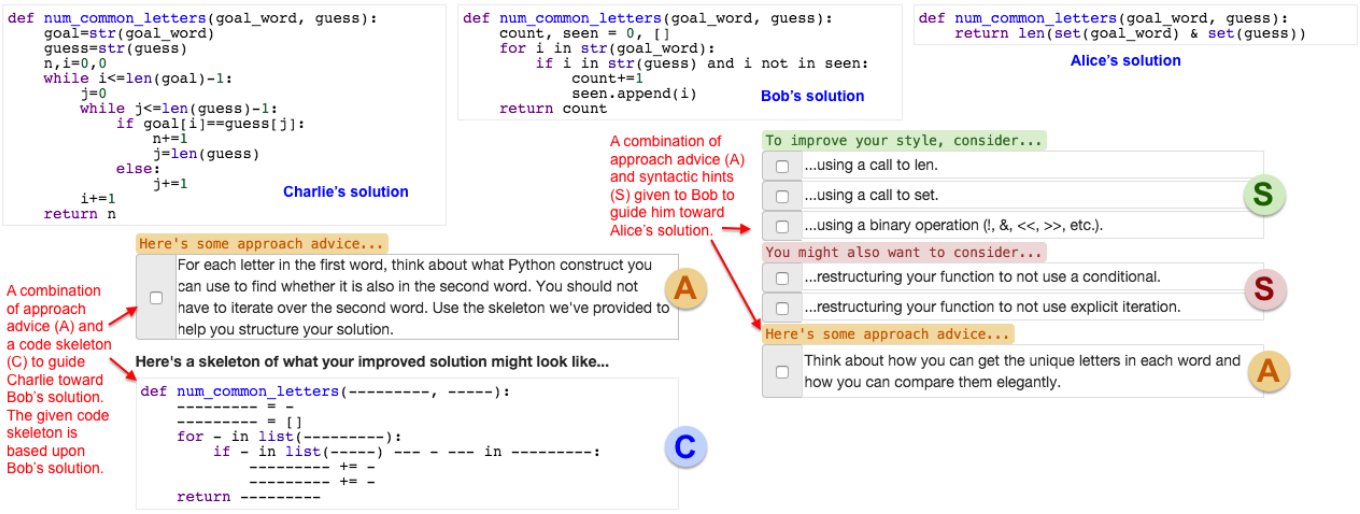


Figure 4: Example of a chain and the hints generated for such a chain.

CHAIN-BUILDING

A key component of our hint-generation system is *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.

Syntactic hints are generated by analyzing the path to determine the most important syntactic hints corresponding to structural features present or absent in later links in the chain. The feature vectors used in this analysis are composed of binary values indicating the presence or absence of specific language features such as built-in functions, language idioms, and control flow constructs in each language; we have constructed feature vectors for Ruby, Java, and Python.

We define the importance of a syntactic hint as the likelihood that it would call attention to the most significant stylistic deficiency in the student's submission. Syntactic hints correspond to features in the above-mentioned feature vectors. We frame the problem of determining which features to use as hints as a classification problem solved using a perceptron [16].

We construct an entire chain from the student's submission to one of the best-style solutions, instead of simply looking at submissions similar to the student's but slightly better. By constructing an entire chain we can identify features that not only *appear* in a better solution but also *persist* – i.e. persistently appear in multiple submissions later in the chain. These features are more likely to be relevant to solving a particular problem with good style than those ones that appear only in the immediate next submission in the chain. Using these features allows the perceptron to learn to prioritize hints that are based on features that don't appear in the student's submission but persistently appear in solutions later in the chain.

Figure 4 is an example of a small chain built by our system for a programming assignment that involves finding the number of common letters between two words in Python. We can see how the hints given to Bob—to use a call to `set()` and `len()` and to use a binary operator—have been derived from the next solution along the chain – Alice's.

Our instructor interface allows instructors to view chains starting from any submission in the graph. The instructor can provide feedback about which hints are not helpful. The system learns from this feedback by adjusting the perceptron weights, which in turn adjusts the hints.

EXPERIMENT 1: 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] The recruited participants were from UC Berkeley's large-enrollment introductory computer science course, CS 61A, which introduces a range of programming concepts, primarily 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

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

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 instructed 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.

EXPERIMENT 1: RESULTS

We collected every correct submission made during the experiment for both groups. Figure 6 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 6 and 7 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. 7, 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 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 $A + C$).

EXPERIMENT 2: DESIGN AND SETUP

We performed another intervention experiment using $n = 145$ student participants to further evaluate the effectiveness of each of the different hint types generated by our system by isolating their effects.[2] The intervention was carried out as an optional extra-credit assignment given to students in UC Berkeley's software engineering course (CS 169), which is primarily taught in Ruby and is composed of third or fourth year undergraduates who have substantial programming experience.

The goal of this experiment was to observe the effects of each of the different types of hints on students in each of the different clusters and see if certain hint types are more helpful to students in certain clusters. The hypothesis is that skeleton and approach hints are more helpful to students in `weak` or `average` clusters since they explain/demonstrate concepts while syntactic hints are more useful to students in `good` clusters since they refer

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

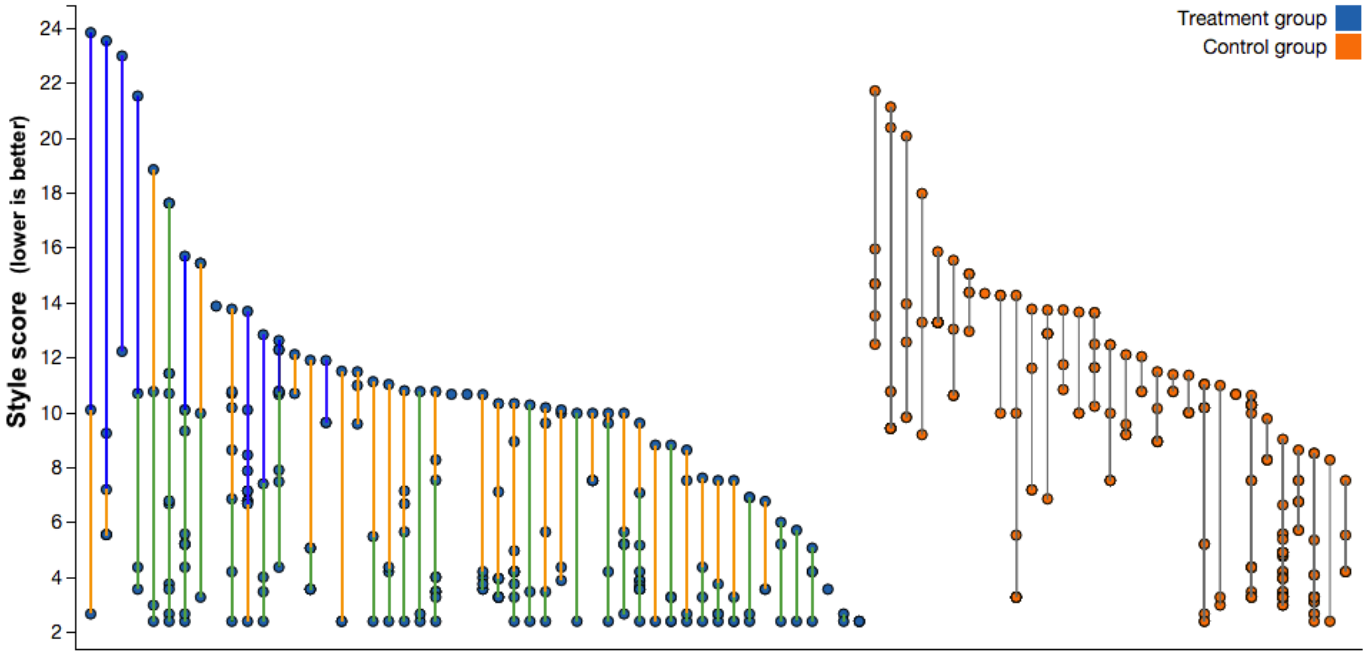


Figure 6: Each vertical line represents a student and each dot along the line is a submission. The color of line segments between dots for the treatment group codifies the combination of hints the student received— blue: `approach` + `code skeleton`, yellow: `approach` + `syntactic`, green: `syntactic` 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 ($p < 0.001$)‡

Figure 7: Key results. †Fisher's exact test ‡Kruskal-Wallis H test

to specific library calls and functions and can help students in good clusters effectively fine-tune their solution.

We selected a Ruby assignment for which we had a corpus of 425 student submissions from a previous offering of the course. Prior to the study, we ran our clustering algorithm and labeled each generated cluster as `good`, `average`, or `weak`; we annotated *all* the clusters with *approach hints*, and picked *exemplars* for *all* the clusters.

All participants were shown the same problem and instructed to submit a solution. 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, the best possible style score for this problem (6.8 based on the corpus of previous submissions—recall that lower ABC scores are better), and style feedback. They were asked to revise their submission using the provided feedback to work towards the best score. Participants were told that in order to get extra-credit they had to either (a) achieve the best possible style score or (b) spend at least 45 minutes attempting to improve their style using the system. There was no limit on how long students could use the system for.

In particular, upon submitting for the first time, the student was given either syntactic hints, approach hints, or a code skeleton. The type of hint given was selected randomly. For each submission thereafter, the student was given the same *type* of hint as they received for their first submission *unless* they had received approach hints or a code skeleton and did not change clusters. In this case they were given syntactic hints thereafter. This was done in the interest of fairness since each cluster has only one set of approach hints and one code skeleton and giving the same hint repeatedly would not help the student improve.

EXPERIMENT 2: RESULTS

After filtering out students who achieved the best solution on their first attempt (and thus did not receive any hints), we split the remaining students into three groups: (1) students who started out with syntactic hints, (2) students who started out with approach hints, and (3) students who started out with code skeletons. The percentage of students who started out with each type of hint was roughly equal (35% syntactic hints, 32% ap-

proach hints, 33% code skeletons). The distribution of flog scores for students in each of the three groups was also roughly equal (21 ± 10 for syntactic hints, 20 ± 10 for approach hints, 21 ± 13 for code skeletons).

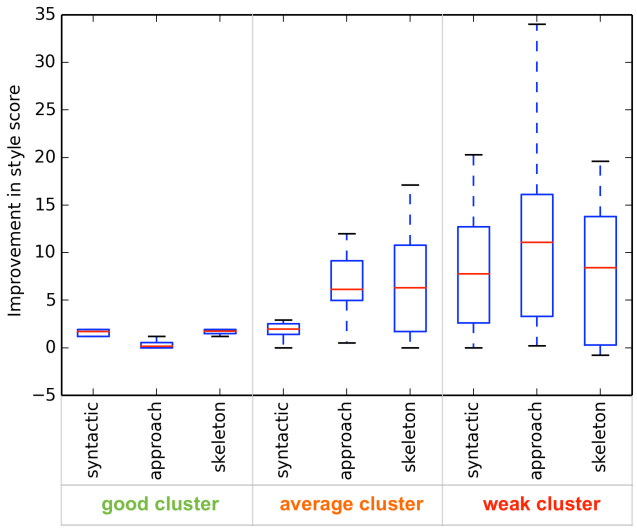


Figure 8: Improvement in style score between first and second submission, separated by initial cluster and type of hint received.

We analyzed the improvement in style score for students in receiving each type of hint in the context of the cluster in which they started out. Figure 8 shows the effect of the types of hints on students who started out in different clusters. The impact of approach hints and code skeletons on students in average clusters appears to be significantly greater than that of syntactic hints. This is further supported by Figs. 9a, 9b. This is as expected since average clusters contain solutions that solve the problem using a mundane approach. Since they advise students on how to approach the problem differently, approach hints are effective here. Code skeletons are similarly effective as they explicitly demonstrate a better approach.

For students who started out in the good cluster, syntactic hints and code skeletons were equally helpful. The relatively low level of improvement shown in Fig. 8 is because students in the good cluster were already close to the best solution and were able to reach the best solution with minor changes. Approach hints were evidently less effective for helpful for students in the good cluster. This is as expected, since students in the good cluster are generally already employing the best strategy and thus do not require approach advice.

For students who started out in the weak cluster, approach hints are more effective that syntactic hints, as can be seen in Fig. 8 and Figs. 9b, 9c. Again, this in intuitive since syntactic hints are likely too specific or fine-grained to significantly improve submissions in weak clusters. It is interesting to see that approach hints were also more effective here than code skeletons. We speculate that this is because approach hints are able to fill conceptual gaps by explicitly describing an approach, while code skeletons have the additional challenge of having to parse the code in the skeleton and attempt to understand the approach it is trying to convey.

Overall, it is evident from this experiment that the type of cluster that a student submission falls into should play a role in determining the type of hint or combination of hints given to the student. Syntactic hints are perceived to be most effective for submissions falling in a good cluster, approach hints and code skeletons are best for submissions in an average cluster, and approach hints are most effective for those in a weak cluster.

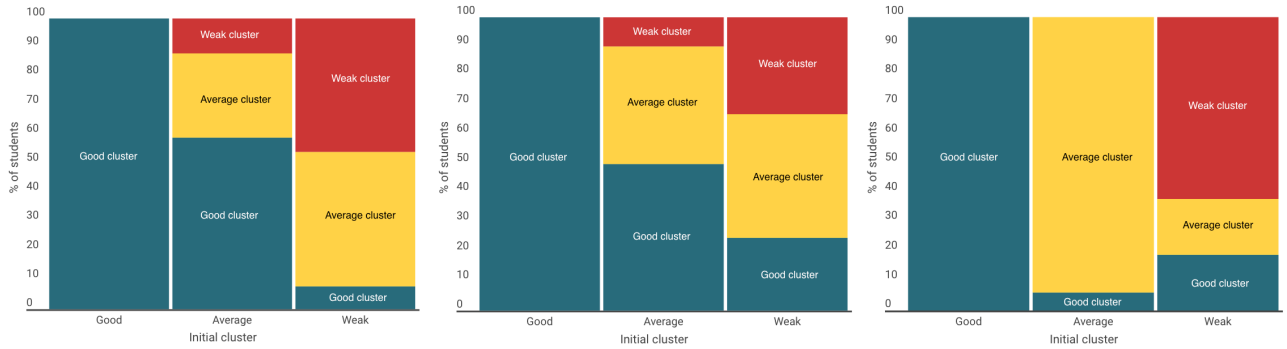
DISCUSSION, LIMITATIONS, ASSUMPTIONS

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. Finally, 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.

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



(a) Code Skeletons (b) Approach hints (c) Syntactic hints

Figure 9: Impact of different types of hints on students who started out in different clusters. Specifically, shows the cluster that students ended up in upon being given a particular type of hint, plotted with reference to their initial cluster.

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 [15], 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 10 shows that this is indeed the case for seven such assignments we studied.

Number of students	Number of Clusters
265	8
425	3
448	5
686	5
951	3
986	6
1607	4

Figure 10: Number of students who submitted a solution to an assignment vs. number of clusters for that assignment for seven comparable assignments.

CONCLUSION

Ultimately, while AutoStyle has its limitations (single function, particular definition of style), experimental deployments in large computer science courses at UC Berkeley have shown the viability and effectiveness of such a system in classroom settings.

Experiment 1 demonstrated that AutoStyle is able to effectively help students improve the quality of their solution to a problem. Students using AutoStyle improved their code style more than those in the control group in a given time frame and the improvement was statistically significant. Moreover, 70% of students using it 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.

Experiment 2 demonstrated that all three types of feedback generated by AutoStyle are relevant and effective, but highlighted that certain types of feedback are more effective under certain circumstances. It showed that syntactic hints are more effective for students whose initial submissions fall in a good cluster, approach hints and code skeletons are best for submissions falling in an average cluster, and approach hints are most effective for those in a weak cluster.

Instructor effort required to use AutoStyle is independent of class size. Thus, integrating systems such as AutoStyle into computer science courses, especially MOOCs, can help future programmers and software developers understand the importance of good coding style and develop good coding style habits early. Moreover, it can serve as an effective way to leverage scale to assist not only students, but also instructors by summarizing the variety of approaches used by students (through clustering) and saving significant instructor time by eliminating the labor-intensive task of manually providing code style feedback.

FUTURE WORK

We plan to field-test AutoStyle 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. K. Ericsson, R. Krampe, and C. Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological review* 100, 3 (1993), 363–406.
2. E. Fast, D. Steffee, L. Wang, J Brandt, and M. Bernstein. 2014. Emergent, Crowd-scale Programming Practice in the IDE. In *SIGCHI Conference on Human Factors in Computing Systems*. Toronto, Canada.
3. J. Fitzpatrick. 2000. Applying the ABC Metric to C, C++, and Java. In *More C++ Gems*. Cambridge University Press, New York, NY, 245–264.
4. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
5. A. Fox and D. Patterson. 2014. *Engineering Software as a Service*. Strawberry Canyon LLC, San Francisco, CA.
6. A. Fox, S. Patterson, D. Joseph, and P. McCulloch. 2015. 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.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
8. E. Glassman, R. Singh, and R. Miller. 2014. Feature Engineering for Clustering Student Solutions. In *1st ACM Conference on Learning at Scale*. Atlanta, GA.
9. T. R. Guskey. 2007. Closing achievement gaps: revisiting Benjamin S. Blooms "Learning for Mastery". *Journal of Advanced Academics* 19, 1 (2007), 8–31.
10. J. Huang, C. Piech, A. Nguyen, and L. Guibas. 2013. 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.
11. S. Johnson. 1977. *Lint, a C program checker*. Technical Report 65. Bell Labs.
12. Stephen H. Kan. 2002. *Metrics and Models in Software Quality Engineering* (2nd ed.). Addison-Wesley, Boston, MA.
13. C. E. Kulkarni, M. S. Bernstein, and S. R. Klemmer. 2015. PeerStudio: Rapid Peer Feedback Emphasizes Revision and Improves Performance. In *2nd ACM Conference on Learning at Scale*. Vancouver, Canada.
14. T. Lazar and I. Bratko. 2014. Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In *Intelligent Tutoring Systems*. Springer, 306–311.
15. R. C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.
16. Warren S McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5, 4 (1943), 115–133.
17. J. Moghadam, R. Roy Choudhury, H. Yin, and A. Fox. 2015. AutoStyle: Toward Coding Style Feedback At Scale. In *2nd ACM Conference on Learning at Scale*. Vancouver, Canada.
18. A. Nguyen, C. Piech, J. Huang, and L. Guibas. 2014. Codewebs: Scalable Code Search for MOOCs. In *23rd International Conference on world wide web*. Seoul, Korea.
19. K. Rivers and K. R. Koedinger. 2015. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* (2015), 1–28.
20. Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53 – 65.
21. V. J. Shute. 2008. Focus on Formative Feedback. *Review of educational research* 78, 1 (2008), 153–189.
22. L. Van der Maaten and G. Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9, 2579-2605 (2008), 85.
23. H. Yin, J. Moghadam, and A. Fox. 2015. Clustering student programming assignments to multiply instructor leverage. In *2nd ACM Conference on Learning at Scale*. Vancouver, Canada.
24. K. Zhang and D. Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.