Autograding for Style

Armando Fox, Hezheng Yin, Rohan Choudhury, Joseph Moghadam

jmoghadam@berkeley.edu

Abstract

1 Introduction

Massive Open Online Courses (MOOCs) provide a powerful way to teach large numbers of students simultaneously. UC Berkeley's course CS 169x is a MOOC whose subject is teaching fundamentals of software engineering. Portions of this course involve students completing programming assignments in ruby. Students receive autograded feedback on the correctness of their submission. However, learning fundamentals of software engineering requires learning to code with good style in addition to coding for correctness. We would like to provide students with automated feedback to help them learn to improve the style of their submission.

Good style in coding is made up of several dimensions such as readability, complexity, modularity, elegance, and even efficiency. There are already automated tools for measuring some aspects of style. For example, flog is a tool that measures ABC complexity (assignments, branches, conditions) and other factors of ruby code. flay is a tool that measures redundant structures in ruby code. However, while these tools do a good job of pointing out problems in code, they do not do a good job teaching solutions. Our goal is not simply to alert students to problems in their code but to teach them concepts that can help them improve their style.

Our tool takes a student submission and automatically reports a hint for a concept the student could apply to improve their style. Instructors and students are able to provide feedback as to whether the hint is useful; our tool is able to learn from this feedback and adjust the hints it gives.

2 Choosing Hints

Set set of hints our tool can give must be predefined, but they can be arbitrary features of code. In our experiments, the features we used were the use or lack of use certain library calls or local groupings of library calls, and the use or lack of use certain data structures. Our intuition for these hints is that many solutions have excessive amounts of code because students failed to realize there already exist library functions with the same functionality.

To give a hint, our tool must decide which of these features to suggest. This is framed as a classification problem and solved with a perceptron; for a particular solution, each possible hint is classified as useful or not based on features about that hint and the particular solution. The most useful hints are reported. The novelty of our tool lies primarily in the design of the features of hints that the perceptron considers.

2.1 Features of a Hint

In order to explain the features our perceptron considers, we must first consider what makes a useful hint.

The usefulness of a hint depends on the particular programming assignment. In a given assignment, there are certain concepts a student should apply in order to attain a solution with good style. For example, one assignment in CS 169x can be written far more cleanly if students understand how to use ruby hashes and do not rely exclusively on arrays. However, this concept may not be applicable for all types of problems.

Furthermore, even within a particular programming assignment, the same hint should not be given to every student. For a given assignment, there is a wide varient of student submissions which form a gradient style, with some submissions having very poor style, some having good style, and some with middling style.

Hence, in order to decide what makes a good hint, we must know what concepts are needed for good style in the particular assignment, and we must know how far off the students we want to advise is from the best possible style.

We propose the following thought experiment: consider a solution with very poor style, and a solution with good style. Then imagine there are small incremental transformation we can make to the poor submission such that it becomes like the solution with good style. For example:

[insert example here]

Now the hint that we should give the poor solution is obvious: it should be a hint that helps it improve one step in the chain. In other words, we should suggest the use of some feature that appears in later submission in the chain and not in the start submission.

The key insight is that these chains of incremental improvements are not confined to a thought experiment; since the class is a MOOC and we have such a large number and variety of students submissions, we can generate these chains by picking out submissions we have received. We can generate a successor link in the chain by finding a submission that is similar to the current submission but improved in style according to a measure of style. In particular, we find the the submission that is closest the tree edit distance of the submissions abstract syntax trees such that the successor submission is lower in flog score, flay score, or run time to some threshold. In principle, this could easily be extended to any measure of similarity and any measure of style.

In summary, we take advantage of the large number of submissions we have received to understand how submissions can improve their style incrementally until they reach the best style; this allows us to provide hints to improve style incrementally.

Specifically, the features associated with each possible hint is whether or not it appears in the current submission, whether or not it appears in the next submission in a chain, whether or not it appears two submissions later in the chain, three submissions later in the chain, etc. This allows the perceptron to learn, for example, that it should prefer hints of concepts that do not appear in the current submission but do appear in later submissions in the chain. We initialize the perceptron to already know this, though it can learn more refined behavior through the feeback it receives.

3 User Interface

We implemented a GUI that can visualize the chain associated with each submission and the hints that would be given for each submission in the chain. This GUI is to benefit the instructor; it would not be shown to students because it would give away the entire solution on how to improve style; we only mean to give one or a couple of hints to each student. However, it can benefit the instructor to see the entire chain at once to be able to give feedback about why hints are helpful or not. Furthermore, we hope that an instructor can gain insight into the variance of student solutions by observing the chains.

The instructor can also change various hyperparameters of the chains. For example, the instructor can move a slider to adjust how similar successive submissions in the chain must be. Closer submissions mean the hints will suggest need smaller changes in code to implement, while further submissions mean the hints will require larger changes in code to implement.

4 Preliminary Experiments

We have run our code on a ruby assignment from CS 169.x and a python assignment from UC Berkeley's CS 61A.

5 Future Work

So far we have experimented with our tool based on data we have collected from previous offerings of CS 169x and CS 61A. We next hope to run our tool in the next live offerings of the course and receive live feedback on its performance.

We also realize that there is a lot of opportunity for expanding the features we consider when classifying hints, or what metrics we use for scoring style. In addition, we wonder if we can improve the chain generation step to account for hint usefulness. Currently, our tool picks which hints it believes are useful based on the chains it generates. It may be possible in turn to simultaneously generate chains in order to highlight useful hints; however, we have not yet decided on an exact mechanism for this.

References