Clustering Student Programming Assignments to Multiply Instructor Leverage

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

Joseph Moghadam
UC Berkeley
jmoghadam@berkeley.edu

Armando Fox
UC Berkeley
fox@berkeley.edu

L@S 2015, March 14–18, 2015, Vancouver, BC, Canada.
ACM 978-1-4503-3411-2/15/03.
http://dx.doi.org/10.1145/2724660.2728695

Author Keywords
clustering; MOOCs; autograding

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

Abstract
A challenge in introductory and intermediate programming courses is understanding how students approached solving a particular programming problem, in order to provide feedback on how they might improve. In both Massive Open Online Courses (MOOCs) and large residential courses, such feedback is difficult to provide for each student individually. To multiply the instructor's leverage, we would like to group student submissions according to the general problem-solving strategy they used, as the first stage of a "feedback pipeline". We describe ongoing explorations of a variety of clustering algorithms and similarity metrics using a corpus of over 800 student submissions to a simple programming assignment from a programming MOOC. We find that for a majority of submissions, it is possible to automatically create clusters such that an instructor "eyeballing" some representative submissions from each cluster can readily describe qualitatively what the common elements are in student submissions in that cluster. This information can

be the basis for feedback to the students or for comparing one group of students' approach with another's.

Introduction

The enormous growth in demand for computing education [8] has resulted not only in a proliferation of Massive Open Online Courses (MOOCs) about programming, but to record enrollments in campus courses: the introductory programming courses at Berkeley and Harvard are now the largest-enrollment undergraduate courses on campus in any department, serving thousands of students per year. A key challenge of helping students learn the computational thinking skills associated with programming is understanding the different problem-solving approaches students used. However, extracting such a high-level description from student source code can be time-consuming, and at large scale, it is impossible for instructors to personally sift through all the solutions.

We would like the instructor to be able to explore and get a comprehensive understanding of these variation easily. In this paper, we propose a clustering based approach which takes advantage of the observation that the diversity of algorithms and language constructs is much smaller than the total number of unique submissions for an open-ended programming assignment. By clustering against well-designed features, we group thousands of submissions into a dozen of "meaningful" clusters. Our qualitative definition of "meaningful" is that an instructor who views a few representative examples from a cluster can readily describe what strategy or elements of the problem-solving approach are exemplified by that cluster.

Clustering and Visualization

Our test corpus consists of 800 student-submitted solutions to a simple "warm-up" programming assignment in the MOOC "Engineering Software as a Service" from UC Berkeley and edX (`http://saas-class.org`): Given an array of strings, group the strings so that all strings in a group are anagrams of each other. Figure 2 shows four examples of actual student submissions, illustrating the wide range of both idiomatic mastery and problem-solving approaches.

In general, there are two different ways to cluster. One is just using feature vectors and another is using a similarity measure between items to be clustered. We did experiemnt in both clustering by features and by similarity metric, but we will focus on the latter in this paper. Since visualizing clusters formed in high-dimensional spaces is challenging, we must do some dimensionality reduction to provide an instructor-friendly visualization. So we have three design choices to make: how to compute similarity between pairs of student submissions; which clustering algorithm to use and how to evaluate the quality of clusters; and how to visualize the clustering results.

Similarity Metric

A natural representation of the syntactic structure of a block of code is its abstract syntax tree (AST), so we need metrics that capture similarity between ASTs corresponding to different student solutions. One widely used metric is the tree edit distance (TED), defined as the minimum cost sequence of node edit operations that transforms one tree into the other. However, instead of using TED directly, we propose a variant of standard TED, which we call normalized TED. Inspired by the top-down structure characteristic of programming, normalized TED emphasizes the importance of high-level

Figure 1: t-SNE visualization of running OPTICS against normalized TED of AST feature (preprocessed by spectral clustering). Each color in the picture represents a cluster. Blue represents outlier points. Silhouette score is 0.545.

program structure by assigning heavier weight to nodes higher up (closer to the root) in the AST. The idea is to prevent minor differences in syntax (that is, code at the leaves of the tree) from unduly affecting the similarity score of programs that are structurally similar but differ in low-level details. Empirically, we find that normalized TED outperforms standard TED in clustering according to the clustering quality score we describe next. A dynamic-programming algorithm by Zhang and Shasha [10] is used to compute both TED and normalized TED.

Clustering Algorithm and Quality

We experimented with several well-known clustering algorithms including k-means, weighted k-means, spectral clustering, DBSCAN and OPTICS. Our stated goal is to produce clusters such that a human instructor can readily "write down a label" describing the strategy represented by each cluster after viewing a few cluster members; empirically we found that the best results were achieved by density-based DBSCAN and OPTICS, which intuitively makes sense since clusters of common solutions are much denser than clusters containing convoluted solutions or outliers (very complex solutions that still manage to be functionally correct).

For measuring cluster quality, we chose the silhouette score (http://en.wikipedia.org/wiki/Silhouette_(clustering)), a widely used internal clustering validation measure that validates clustering performance based on the pairwise difference of between- and within-cluster distances. Mathematically, the silhouette score s is given by

$$s = \frac{1}{n}\sum_{i=1}^{n}\frac{b(i)-a(i)}{\max\{a(i),b(i)\}}$$

where $a(i)$ is the average distance between the i'th sample and all other samples within the same cluster; $b(i)$ is the minimum average distance between the i'th sample and any other cluster in which i is not a member. From the above equation, we have $-1 \leq s(i) \leq 1$. A larger silhouette score generally indicates better clustering quality.

Visualization

To generate a two-dimensional visualization of clustering result, we used t-SNE (http://lvdmaaten.github.io/tsne/), a nonlinear dimensionality reduction technique that aims to preserve the local structure of data. An example of t-SNE visualization is shown in Figure 1. Each color in the figure represents a cluster. Blue points are outliers that do not have enough neighbor points to form a cluster. We built a simple interactive GUI in which the instructor can click on each datapoint to view the corresponding submission.

Example

In this section, we present example results obtained from running OPTICS algorithm against normalized TED of AST feature. The visualization of clustering result is shown in Figure 1.

After the clustering, we manually viewed a few randomly-chosen submissions from each cluster and tried to label each cluster according to what strategy or problem-solving approach is exemplifed by the submissions in that cluster. We found that all four clusters are "meaningful" in that we can easily assign such a label to every cluster. Each cluster's label and a representative submission are illustrated in Figure 2.

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

```ruby
def combine_anagrams(words)
  hash = Hash.new
  words.each do |w|
    key = w.downcase.chars.sort.join
    if hash.has_kay? key
      hash[key] += [w]
    else
      hash[key] = [w]
    end
  end
  hash.values
end
```

```ruby
def combine_anagrams(words)
  hash = Hash.new
  words.each do |w|
    key = w.downcase.chars.sort.join
    if hash.has_kay? key
      hash[key] << w
    else
      hash[key] = [w]
    end
  end
  result = []
  hash.each_key do |key|
    result << hash[key]
  end
end
```

```ruby
def combine_anagrams(words)
  result = []
  words.each do |w1|
    temp = []
    words.each do |w2|
      if (w1.downcase.chars.sort.join
          == w2.downcase.chars.sort.join)
        temp << w2
      end
      if !result.include?(temp)
        result << temp
      end
    end
  end
  return result
end
```

Figure 2: These four solutions are from red, purple, orange and green clusters respectively. Red: a canonical solution with average solution length of 3.7 lines; Purple: a typical but longer solution, average solution length 10.9 lines; Orange: similar to Purple, but students are unaware of the `values` method available to collect all the values in a hash table, resulting in longer average solution length of 12.5; Green: complex solutions with average length 21.3 lines.

We observed that red, purple and orange clusters have less internal variation than green (complicated solution) cluster, which intuitively make sense since there are many more ways of writing a complex solution than of writing a simpler one.

To illustrate the benefit of normalized TED, we compared it with raw TED by running OPTICS against these two features. The reachability plots obtained from OPTICS are shown in Figure 3 and Figure 4.

Figure 3: Reachability plot of normalized TED generated by ELKI, yellow represents outliers

Figure 4: Reachability plot of standard TED generated by ELKI, orange represents outliers

For OPTICS, the clusters show up as valleys in the reachability plot. A deeper valley usually indicates a denser cluster. From Figure 3 and Figure 4, we can see that normalized TED has more stable clusters and fewer outliers than TED.

Related Work

We are inspired by early work on feature engineering for automatic clustering of student programs [3], which proposed the use of variation theory to explain differences in student submissions. Applying variation theory to programming assignments essentially means using a hierarchical approach: first, distinguishing clusters of approaches to solving a problem, and then within each cluster identifying differences in various students' implementation of a particular approach.

This "divide and conquer" approach has many other use cases, including pairing students based on their problem-solving strategies and making it easier to illustrate to students the relateive merits of different approaches. For the moment our main focus is to multiply instructor leverage rather than fully automating the feedback process. We envision, for example, that our techniques might be used in conjunction with a system such as "Divide and Correct" for providing feedback on large groups of student submissions at once [1], or embodied in a system similar to OverCode [9], which helps visualize major groupings of variation among student submissions to a code assignment.

We use Abstract Syntax Trees (ASTs) as the basis of representing program structure and explore various metrics of similarity between ASTs. Much previous work has used ASTs to detect plagiarism [4, 6] or flag needless code duplication in a codebase by using techniques such as Latent Semantic Analysis to compare source files [5]. As in our approach, most previous work focuses on the fundamental difficulty that "raw" AST differencing captures differences at too low a level of abstraction to permit identifying higher-level patterns in the code. More recently, approaches such as Codewebs [7] compare "code phrases," or subgraphs of ASTs, to determine the probabilistic semantic equivalence of two pieces of code. An alternative approach is to provide an instructor with triples $R, C1, C2$ of solutions and ask which of $C1$ or $C2$ is more similar to R in the instructor's opinion [2], then evaluate several algorithms according to which ones show highest agreement with the instructor's opinion. This approach revealed Jaccard distance and various term-frequency based distances as promising candidates for AST differencing, but was tried only on synthetic data consisting of mutated reference solutions, not real student data.

Acknowledgments

Thanks to John Canny for valuable feedback and advice on clustering metrics and the project as a whole. This work was supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation

of China Grant 61033001, 61361136003.

References

[1] Brooks, M., Basu, S., Jacobs, C., and Vanderwende, L. Divide and correct: Using clusters to grade short answers at scale. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, ACM (New York, NY, USA, 2014), 89–98.

[2] Gaudencio, M., Dantas, A., and Guerrero, D. D. Can computers compare student code solutions as well as teachers? In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, ACM (New York, NY, USA, 2014), 21–26.

[3] 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.

[4] Lancaster, T., and Culwin, F. A Comparison of Source Code Plagiarism Detection Engines. *Computer Science Education 14* (June 2004), 101–112.

[5] Maletic, J., and Marcus, A. Supporting program comprehension using semantic and structural information. In *23rd Intl. Conf. on Software Engineering (ICSE)* (Toronto, Canada, May 2001).

[6] Martins, V. T., Fonte, D., Henriques, P. R., and da Cruz, D. Plagiarism detection: A tool survey and comparison. *OpenAccess Series in Informatics* (2014).

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

[8] Patterson, D., and Lazowska, E. Why are English majors studying computer science? http://blogs.berkeley.edu/2013/11/26/why-are-english-majors-studying-computer-science/, Nov 2013.

[9] 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).

[10] 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.