Feature Engineering for Clustering Student Solutions

Elena L. Glassman **Rishabh Singh** **Robert C. Miller**
MIT CSAIL, 32 Vassar St. Cambridge, MA
{elg,rishabhs,rcm}@mit.edu

ABSTRACT
Open-ended homework problems such as coding assignments give students a broad range of freedom for the design of solutions. We aim to use the diversity in correct solutions to enhance student learning by automatically suggesting alternate solutions. Our approach is to perform a two-level hierarchical clustering of student solutions to first partition them based on the choice of algorithm and then partition solutions implementing the same algorithm based on low-level implementation details. Our initial investigations in domains of introductory programming and computer architecture demonstrate that we need two different classes of features to perform effective clustering at the two levels, namely *abstract* features and *concrete* features.

Author Keywords
Algorithm recognition; program comprehension; feature engineering

ACM Classification Keywords
K.3.1 Computers and Education: Computer Uses in Education—Computer-assisted instruction (CAI)

INTRODUCTION
There are a variety of ways in which students implement solutions for open-ended homework problems such as coding assignments. Their correct solutions vary in at least two dimensions: (i) choice of algorithm, and (ii) choice of language constructs and library functions for the low-level implementation. This variation among correct solutions gives us an opportunity to use them to enhance student learning, in accordance with Marton et al.'s Variation Theory (VT) [3]. VT holds that in order to learn concepts, one must see examples that vary along dimensions of *contrast*, *generalization*, *separation*, and *fusion*. In this work, we aim to build a system that can automatically provide students with examples of alternative correct solutions across these different dimensions, *powered by a large dataset of previous student solutions*.

In order to separate solutions along VT's recommended dimensions, we must design metrics that capture the distinctions VT makes between solutions. Our first exploratory feature design study is based on a large dataset of students' Python submissions from an introductory programming course offered on the edX MOOC platform in Fall 2012.

We show a few hand-picked examples in Figure 1 from the `comp-deriv` problem, which computes the derivative of a polynomial. To illustrate VT's contrast dimension, we include an example of a `comp-deriv` solution paired with a solution to a different problem, `eval-poly`. Under the generalization heading, we have shown two solutions that use the same approach or algorithm, but different low-level functions and language constructs to implement it. We illustrate the separation dimension of variation by pairing two `comp-deriv` solutions that implement different algorithms.

RELATED WORK
A common goal of the prior work cited here is to help teachers monitor the state of their class, or provide solution-specific feedback to many students. However, the techniques for analyzing solutions have not converged on a particular method. Huang et al. [1] use unit test results and AST edit-distance algorithms to identify clusters of submissions that could potentially receive the same custom feedback message. Taherkhani et al. [4] identify which sorting algorithm a student implemented using supervised machine learning methods. Each solution is represented by statistics about language constructs, measures of complexity, and detected roles of variables.

Luxton-Reilly et al. [2] label types of variations as structural, syntactic, or presentation-related. The structural similarity is captured by the control flow graph of the student solutions. If the control flow of two solutions is the same, then the syntactic variation within the blocks of code are compared by looking at the sequence of token classes. Presentation-based variation, such as variable names and spacing, is only examined when two solutions are structurally and syntactically the same. Our motivation is similar to that of Luxton-Reilly et al., but we explore a less strict notion of solution similarity.

OUR APPROACH
We are pursuing a two-level hierarchical clustering methodology. The high-level clusters are intended to partition solutions along the separation dimension, where each cluster represents a particular algorithm. We have used k-means to create these high-level clusters of solutions based on abstract features. The abstract features for Python programs consist of 12 features that include the position of conditional statements relative to the loop statements (before, after, or inside), the depth of nested loops, number of AST nodes, return statements, loops, comparisons, etc.

The sub-clusters within each high-level cluster are intended to capture the generalization dimension, where the only dif-

L@S'14, March 4–5, 2014, Atlanta, GA.
ACM 978-1-4503-2669-8/14/03
((DOI HERE))

CONTRAST

```
def computeDeriv(poly):
    ans = []
    for i in range(1,len(poly)):
        ans.append(i*poly[i])
    if ans == []:
        ans = [0.0]
    return ans
```

```
def evaluatePoly(lis,a):
    total = 0
    for i in range(len(lis)):
        e = lis[i]*(a**i)
        total = total + e
    return total
```

GENERALIZATION

```
def computeDeriv(poly):
    powers = len(poly)
    if powers == 1:
        return [0.0]
    deriv = []
    for i in range(powers):
        deriv.append(poly[i]*i)
    return deriv[1:]
```

```
def computeDeriv(poly):
    if len(poly) > 1:
        res = []
    else: return [0.0]
    for i in range(len(poly)):
        res.append(poly[i]*i)
    res.pop(0)
    return res
```

SEPARATION

```
def computeDeriv(poly):
    idx = 1
    res = list([])
    polylen = len(poly)
    if polylen == 1: return [0.0]
    while idx <= polylen:
        coeff = poly.pop(1)
        res.append(coeff*idx)
        idx = idx + 1
        if len(poly) < 2: return res
```

```
def computeDeriv(poly):
    result = []
    for i in range(1,len(poly)):
        result.append(i*poly[i])
    if len(result) == 0:
        result.append(0.0)
    return result
```

Figure 1: Hand-selected examples of student solutions varying along Variation Theory dimensions. Students were asked to implement a function to compute a polynomial's derivative; the polynomial's coefficients are represented as a list. The contrast dimension contains examples that are and are not a derivative-computing function. The generalization dimension includes examples with the same algorithm but different low-level implementations. The separation dimension captures the full variation of implementations which still compute the derivative of a polynomial.

ferences between clusters are low-level language constructs and used library functions. We plan to use k-means again on solutions within each high-level cluster, based on low-level, concrete features. The concrete features for Python programs consist of 48 low-level features that include the number of specific types of operators (add, subtract, etc.), comparisons ($<$, $>$, etc.), loops (while or for), library functions, and statements (assignments, conditional, or loop), number of program variables, constant values, etc.

PRELIMINARY RESULTS

We use abstract features for k-means clustering of student solutions for the separation dimension, which partitions the solutions into k clusters. We compute clusterings for different k values, and then compare these clusterings to those created by two course teaching assistants (TAs). The TAs were given 50 randomly chosen student solutions as a clustering task. We did not give them specific directions for clustering, in order to better understand how the TAs naturally group solutions. We observed that they ignored low-level features, e.g., they clustered together solutions implementing the same algorithm but using different functions such as `pop`, list slicing, and `delete`.

We use the adjusted mutual information (AMI) metric to compare TAs' clusterings with each other and with our k-means clustering. An AMI value of 0 indicates purely independent clusterings, whereas a value of 1 indicates perfect agreement between the clusterings. The agreement of the two TAs' clusterings, referred to here as the inter-TA AMI, is only 0.3275. When k was sufficiently high, i.e., at least 15, the k-means-produced clusterings agreed, as measured by AMI, with each TA's clusterings as much or more than the TAs' clusterings agreed with each other. We found high agreement between our k-means and TA-produced clusterings on two additional coding assignments as well.

FUTURE WORK

We are generalizing this approach to two additional domains. The Mathworks runs an online game, Cody. Users submitted 218,000 Matlab functions as solutions to 1000 or so problems. We hope to categorize software metrics, library functions, and language constructs within Matlab functions as abstract features, differentiating algorithms, or concrete features, distinguishing implementations of the same algorithm. The second domain is code written by MIT students in a hardware description language. Students define their own library of circuits, from which larger circuits are composed. Within each high-level cluster based on overall structure, we could cluster based on low-level library circuit implementation.

REFERENCES

1. Huang, J., Piech, C., Nguyen, A., and Guibas, L. J. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED Workshops* (2013).
2. Luxton-Reilly, A., Denny, P., Kirk, D., Tempero, E., and Yu, S.-Y. On the differences between correct student solutions. In *ITiCSE '13*, ACM (2013), 177–182.
3. Marton, F., Tsui, A., Chik, P., Ko, P., and Lo, M. *Classroom Discourse and the Space of Learning*. Taylor & Francis, 2013.
4. Taherkhani, A., Korhonen, A., and Malmi, L. Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In *Koli Calling*, ACM (2012), 83–92.