Chapter 1

~~Extensions of OverCode~~ Bayesian Clustering of Student Solutions

The original vision of OverCode was to discover pedagogically valuable themes of variation within thousands of student solutions to the same programming ~~exercise. While OverCode does canonicalize and cluster correct solutions so that they can be more easily understood as a group, it does not pull out larger themes of variation as clearly as initially hoped for, nor does it handle incorrect solutions . This chapter describes more recent efforts to address both these shortcomings.~~

~~One approach to pulling out larger themes of variation within solutionsis to cluster more aggressively. OverCode's existing~~ problem. Each Python solution is normalized before clustering and represented as a sequence of normalized lines of code. All solutions that share the same set of normalized lines of code are clustered together. The clustering is a strict partition of the solutions. This clustering process is hard and agglomerative but not hierarchical. It was designed with interpretability in mind. Hierarchical clustering may also be appropriate in this context, as long as human interpretability does not suffer. The OverCode clustering pipeline is ~~a form of deterministic interpretable clustering. What~~ also interactive, because the results are shown in an interface that allows the human to tell the system that some lines of code are not different in meaningful ways. If any clusters differ

only by those lines of code that the human considers to be the same, those clusters will be merged.

In OverCode, what can vary and what is invariant across all the solutions within a ~~cluster~~ stack is clear, just by reading the ~~canonicalized~~ platonic solution that represents the ~~cluster. Because it faithfully represents the syntax students used in each cluster, solutions are split apart into smaller clusters by small syntactic differences.~~

~~One promising set of methods for more aggressive groupings of solutions are bottom-up and rule-based: they define rules for which solutions or~~ stack. Syntactic differences split apart groups of solutions ~~can be merged into a single cluster. Probablistic semantic equality or equality with respect to a teacher's test suite, which is already used by OverCode for variable names, can be used for subexpressions as well []. Programming language-based methods, e.g., compiler optimizations, could collapse OverCode clusters based on known semantic equality. As a system designer, one must decide or give the teacher control over how many different, semantically equivalent solutions are collapsed into a single cluster since, at the highest level, all correct solutions to the same programming exercise are semantically equivalent. These methods are discussed again in the future work, because the second set of methods were chosen for future investigation in this chapter~~into smaller stacks so that each stack's platonic solution matches the syntax used in the solutions in represents. This can also produce more stacks than teachers can read. OverCode does normalize and cluster correct solutions so that they can be more easily understood as a group, but it does not pull out larger themes of variation as clearly as initially hoped.

~~The second set of methods leverage statistical properties of the entire corpus of solutions~~

However, OverCode's normalized stacks do exhibit statistical regularity. Given that ~~Variation Theory~~ variation theory is interested in dimensions of variation ~~(and consistency)~~ and consistency that characterize all possible instantiations of an idea, statistical methods warrant further investigation. Latent variable models are a type of statistical model that attempt to explain variation in a dataset based on underlying factors. With the right choice of features and model, a latent variable model may be able to capture underlying design choices. In

this section, two latent variable models have been explored in a preliminary way for this purpose: the Bayesian Case Model (BCM) [citation removed from diff] and Latent Dirichlet Allocation (LDA) [citation removed from diff].

BCM was selected because, like the original OverCode pipeline, it clusters solutions and produces a single solution to represent the entire cluster. Like OverCode, it also indicates the features that characterize each cluster. While OverCode displays the ~~set of canonicalized lines that all members of a clusterdeterministically share~~platonic solution that shares the same set of normalized lines with all other members of the cluster, BCM learns a subspace of features that it determines are most characteristic of the solutions within ~~cluster, in a probabilistic sense. It also~~ that cluster.

BCM has an interactive variant called iBCM~~, which allows the user~~. iBCM allows the teacher to directly modify the prototype and the subspace chosen by BCM if it disagrees with their domain knowledge or preferences. This ~~user modification~~ teacher action triggers a rerun of BCM with the modifications taken into account.

Internally, BCM depends on a mixture model with a Dirichlet prior. Rather than find a cluster for each entire solution, mixture models can learn clusters of features that co-exist across some subset of all solutions. The concentration parameter of ~~BCM's~~ the BCM Dirichlet prior is set to promote sparsity, i.e., mixture distributions over solutions that have the majority of their probability mass on a single mixture component. BCM then assigns each solution to a cluster according to its most probable mixture component.

~~LDA~~ Since teachers can be poor at clustering solutions, i.e., inconsistent with each other [**?**], it may also be difficult for any algorithm to find a clustering that appeals to most teachers. However, there are a statistical models, lie LDA, that cluster components of solutions instead of entire solutions. LDA was selected as an alternative ~~statistical approach to BCM because of evidence collected at UCBerkeley [] that experts were themselves not particularly consistent about how to cluster solutions. While LDA's output is not as optimized for interpretability as BCM 's, it preserves the ability to examine solutions through the lens of their mixture components, rather than clusters. LDA learns both mixture components,~~

~~which are distribution over features, and the distributions of those mixtures over each solution in the set it is trained on. Depending on the featureschosen to represent solutions, it may~~ to BCM in order to explore how solution component clustering might better support teachers who do not agree on whole-solution clusterings.

BCM was built explicitly with interpretability in mind, but it is still not clear whether LDA or BCM are more interpretable in the context of clustering OverCode stacks. LDA has some known interpretability problems. As part of its training process, LDA learns distributions over features, also known as *mixture components*. It can be difficult to interpret exactly what ~~a mixture component is~~ an LDA mixture component means just by looking at its distribution over features. ~~However, sorting solutions based on the degree to which a particular topic is associated with them may pull out a distinctive, human-interpretable themes.~~

1.1 Clustering Solutions with BCM

BCM was applied to the ~~canonicalized cluster-representing~~ platonic solutions generated by OverCode. ~~These canonicalized~~ The platonic solutions encode both static and dynamic information~~in a readable function~~, i.e., the syntax carries the static information and the variable ~~name~~ naming encodes dynamic information. The ~~canonicalized function body~~ platonic solution is tokenized and represented as ~~binary vectors~~ a binary vector indicating the existence of the features, including renamed variables and language-specific keywords, such as ~~specific canonicalized~~ normalized variable names like `listA` and Python keywords like `assert` and `while`. The result is a ~~BCM~~ clustering of OverCode ~~clusters~~platonic solutions.

~~In a small pilot study, three introductory Python teachers were each given sets of Python solutions to three~~

1.1.1 Preliminary Evaluation

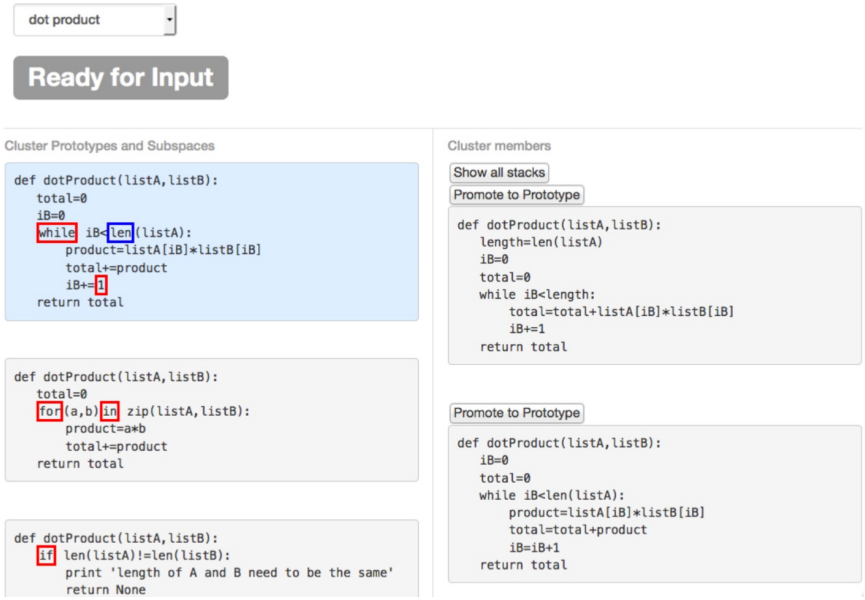

Figure 1-1: This interface displays iBCM clusterings of OverCode platonic solutions and affords user feedback on cluster quality. The solutions on the left are cluster prototypes. The blue solution is the selected cluster whose members are shown in a scrollable list on the right-hand side. The tokens contained in red boxes are features that BCM identifies as characteristic of the cluster represented by that prototype. When a user hovers their cursor over a keyword or variable name, e.g., `len`, it is highlighted in a blue rectangle, indicating that it can be clicked.

BCM was run on three different programming problems selected from those previously analyzed in ~~the chapter on OverCode. For each problem, they were asked to create a grading rubric and provide helpful comments for the students, based on interacting with solutions in one of three different interfaces. The three interfaces were: (1) raw solutions in the browser, similar to~~ Chapter **??**. An interface, shown in Figure **??**, displayed the results, with iBCM continuing to run in the ~~control used in the OverCode studies, (2) OverCode's canonicalized cluster-representating solutions, and (3) a BCM clustering of OverCode clusters, i. e., the prototypes and characteristic features of each BCM cluster, as well as its cluster members. The BCM interface , shown in Figure **??**, was running the iBCM variant of the algorithm, so teachers could promote~~ backend, ready to update the clustering in response to teacher feedback. The interface afforded promoting a member of a cluster to be its prototype ~~or click~~ and clicking on a token within a ~~prototype–a~~ prototype, such as a variable name or ~~keyword–to~~ keyword, to toggle whether or not it is ~~considered~~ labeled a characteristic

feature of that cluster. ~~Both these modifications triggered BCM running in the backend to rerun and send a new clustering to the front end for display~~

In order to determine whether or not these clusterings and interface affordances are useful to teachers, a small pilot study was run on three teachers of introductory Python programming. For the pilot, two more interfaces were introduced as controls. One is a duplicate of the control interface used in the OverCode studies, i.e., a long list of syntax highlighted raw solutions in a random order in the browser. The second interface differs from the previous control by one factor: the raw solutions are replaced with their platonic equivalents.

The teachers were each given sets of Python solutions to view in each of the three interfaces. For each problem, they were asked to create a grading rubric and provide helpful comments for the students of the type that might be mentioned in a class-wide forum post.

~~Pilot users~~

The teachers appreciated the fact that BCM gave some structure to the space of solutions~~; rather~~. Rather than a long list of solutions, the interface suggested distinct subpopulations of solutions within the list. However, subjects did not fully understand the probabilistic nature of the clustering method. The presence of a single ~~"intruder"~~intruder, i.e., a solution that the teacher believed did not belong in a cluster, caused confusion. This could be ameliorated by giving ~~users~~ teachers more ways to modify the clustering, e.g., allowing ~~users~~ teachers the option to kick an intruder out of a cluster and rerun BCM~~, or by introducing the tool as a mechanism for "discovery" instead of "organization. "~~. Subjects also requested richer or higher-level features than variable names and keywords. Been Kim's PhD thesis [**?**] describes a follow-up full user study comparing the efficacy of BCM ~~vs.~~ and iBCM on clustering OverCode ~~cluster-representing~~ platonic solutions.

~~The solutions on the left are cluster prototypes. The blue solution is the selected cluster whose members are shown in a scrollable list on the right hand side. The tokens contained in red boxes are features that BCM "believes" characterize the cluster represented by that prototype. When a user hovers their cursor over a keyword or variable name, e.g.,~~ `len`~~, it~~

~~is highlighted in a blue rectangle, indicating that it can be interacted with, i.e., clicked.~~

1.2 Clustering Solution Components with LDA

~~As described in the chapter on related work, other~~ Other researchers have documented a lack of agreement across human-made clusters of student ~~code~~solutions [**?**]. One possible explanation for ~~this low consistency across teacher-made clusterings of student code~~ poor consistency is that solutions are mixtures of design choices and teachers care ~~about different things. As described in [**?**]: the clusters can be as straight forward as "1 pt", "2 pts" and "3 pts".~~ differently about various aspects of solutions. If student A writes a solution with a well-written loop and extraneous statements while student B writes a solution with extra loops but otherwise very clean code, teachers can reasonably disagree about which cluster each whole solution should be placed in, depending on whether they believe inefficient control flow or extraneous statements are worse.

Instead of trying to approximate clusterings that humans do not even agree on, it may be more useful to model solutions as mixtures of good and bad design choices. While more sophisticated mixture models' assumptions may ultimately be more appropriate, LDA [**?**] as implemented in the Gensim toolbox [**?**] was chosen as the model to evaluate in this preliminary work.

Like BCM, LDA was run on the ~~canonicalized cluster-representing solutions, not raw solutions~~platonic solutions. However, the representation of these solutions was also changed: in order to pull out higher-level patterns in approach, rather than lower-level patterns in syntax, solutions were represented solely by the behavior of the variables within them.

As described in the chapter on OverCode, the OverCode analysis pipeline executes all programs on a common set of test cases and records the sequence of values taken on by each variable in the program. OverCode assumes that variables in different programs that transition through the same sequence of values on the same test cases are in fact fulfilling

the same semantic role in the program.

```
def iterPower(base, exp):
    '''
    base: int or float.
    exp: int >= 0

    returns: int or float, base^exp
    '''
    # Your code here
    if exp <= 0:
        return 1
    else:
        return base * iterPower(base, exp - 1)
```

Figure 1-2: Example of a recursive student solution.

```
def iterPower(base, exp):
    result = 1
    while exp > 0:
        result *= base
        exp -= 1
    return result
```

Figure 1-3: Example of a ~~`while`-based~~ student solution using the Python keyword `while`.

~~Below are~~ Figure **??** shows the sequences of variable values recorded by OverCode while executing `iterPower(5,3)` as defined in Figures **??**, **??**, and **??**:

In ~~the previous~~ these examples, the input argument `base` would be considered a variable common to all three programs, but the input argument `exp` would not be. The variable `result` would also be considered a common variable shared across just the definitions in Figure **??** and Figure **??**. This allows us to distinguish between programs that calculate the answer in semantically distinct ways, without discriminating between the low-level ~~syntax-based design decisions~~ design decisions about syntax.

LDA is often applied to ~~corpuses~~ corpora of textual documents, where the corpus is represented as a $W \times N$ term-by-document matrix of counts, where W is the vocabulary size across all documents and N is the number of documents. In this representation, the document is represented a bag of word counts, i.e., how many times each word appears in the document.

```
def iterPower(base, exp):
    iter = exp
    result = 1
    while iter > 0:
        result = result * base
        iter = iter - 1
    return result
```

Figure 1-4: Example of a ~~while-based~~ student solution using the Python keyword `while`, where the student has not modified any input arguments, i.e., better programming style.

- **Figure ??**
  - `exp: 3, 2, 1, 0, 1, 2, 3`
  - `base: 5`
- **Figure ??**
  - `exp: 3, 2, 1, 0`
  - `base: 5`
  - `result: 1, 5, 25, 125`
- **Figure ??**
  - `exp: 3`
  - `base: 5`
  - `iter: 3, 2, 1, 0`
  - `result: 1, 5, 25, 125`

Figure 1-5: The sequences of variable values recorded by OverCode while executing `iterPower(5,3)`.

Following this analogy, solutions are represented as a bag of variable behaviors. The matrix representing the solutions in Figures **??** through **??** executed on `iterPower(5,3)` is shown in Table **??**.

Note that, while the entries in Table **??** only take on values 0 or 1, more complicated definitions may have n instances of, e.g., a variable that takes on the sequence of values `3,2,1,0`. In that case, there could be an n in the `3,2,1,0` column for the row corresponding to that solution, or it can be left as a binary indicator.

In order to run LDA, ~~3875~~ student solutions to ~~`iterPower`~~ ~~were~~ a particular problem are first run on a set of test cases within the OverCode analysis pipeline. The OverCode pipeline ~~produced~~ produces a set of ~~977 cluster-representing~~ platonic solutions and a set of features for each solution, including which variable sequences were observed during

Table 1.1: Variable-by-Solution Matrix for Programs, where variables are uniquely identified by their sequence of values while run on a set of test case(s)

SOLUTION	5	1,5,... 25,125	3,2,1,0, ...1,2,3	3,2,1,0	3
~~FIG.~~ FIGURE ??	1	0	1	0	0
~~FIG.~~ FIGURE ??	1	1	0	1	0
~~FIG.~~ FIGURE ??	1	1	0	1	1

execution. ~~Another script turned this output into a~~ A script extracts the variable-by-solution matrix~~for the 977 cluster-representing solutions, which were then fed into LDA for analysis. LDA was~~. LDA is then run repeatedly with ~~multiple~~ different values for the parameter that sets the number of latent mixture components and then inspected by hand. The results ~~were examined by hand since~~ of running LDA on platonic solutions is inspected by hand because cluster validity metrics like perplexity and held-out likelihoods are not necessarily good proxies for human interpretability [**?**].

~~Since the learned mixture components~~ Inspection by hand can also be challenging. When LDA learns mixture components in this context, those components are distributions over variable behaviors~~, it is easier to inspect solutions which have high "amounts "~~. Rather than inspect these distributions, entire solutions containing high amounts of that mixture component ~~"within" them and infer a theme by comparing them to solutions that contain high "amounts " of a different mixture component. These comparisons were done by hand for a subset of popular mixture components for each LDA model.~~ were inspected instead. By comparing entire solutions with high and low amounts of each component, one can infer what each learned component captured. This method for understanding the results is similar to the recommendations of variation theory.

1.2.1 Preliminary Evaluation

3875 student solutions to `iterPower` were run on a set of test cases within the OverCode analysis pipeline. The OverCode pipeline produced a set of 977 platonic solutions and sets of features for each solution. LDA was run on the resulting variable-by-solution matrix,

using different parameter values.

The LDA results were inspected by hand. One topic comparison captured the difference between solutions like Figure **??** and **??**. Another topic comparison exposed the difference between the subpopulation of solutions with ~~extra (unnecessary)~~ unnecessary conditional statements and the common, more concise solution. ~~In the future, a user interface would be very helpful for this task, especially one which made it easier to compare the output of models with different parameter values.~~

1.3 Discussion and Future Work

LDA applied to a variable-by-solution matrix is a promising method for identifying variation within ~~corpuses~~ corpora of solutions to the same programming ~~exercise~~problem. However, ~~LDA's assumptions~~ the assumptions made in LDA, such as the independence of mixture components, and requirements, such as explicitly setting the number of mixture components beforehand, may mean that other mixture models, such as the Correlated Topic Model ~~[]~~ [**?**] or the Hierarchical Dirichlet Process ~~[]~~ [**?**] will ultimately be a better model fit for this purpose.

~~While understanding the contents of thousands of correct student solutions can be helpful in both residential and online contexts, another application of OverCode's pipeline and interface is supporting the hand-grading of introductory Python programming exam solutions, only some of which are correct. Incorrect solutions are defined as those that do not pass at least one test case in the teacher-designed test suite.~~

~~Hand-reviewing solutions is necessary because teacher-designed test suites were found to unfairly penalize some students and award undeserved credit to others. For example, a single typo in an otherwise well-written solution can cause it to fail all the test cases and receive no credit. Conversely, a solution that subverts the purpose of the assignment can still receive credit by returning the expected answer to some or all of the test cases.~~

For the staff of 6.0001, the residential introductory Python programming courses at MIT, this can be one of the most time-consuming and exhausting parts of teaching the course. It can take an full workday for the entire staff of eight to ten teachers to sit in a room and review several hundred students' solutions by hand in order to assign a single numerical grade to each solution.

There are two main technical contributions found in the GroverCode extension of OverCode: a modified pipeline that can canonicalize both correct and incorrect solutions and an interface designed for grading. Just as the original pipeline used variable behavior to canonicalize variable names, the modified pipeline uses variable behavior and the syntax of statements containing each variable to canonicalize variable names in solutions that are not correct. This comes from the insight that a variable in an incorrect solution can be semantically equivalent to a variable in a correct solution but still behave differently; it can behave differently due to a bug in a line of code that directly modifies its value or a bug in a line of code that affects the behavior of another variable that it depends on. Since many of the exam solutions are incorrect and do not get clustered together, the new user interface organizes solutions according to their behavior on test cases and compositional similarity to each other, rather than by cluster size.

GroverCode was iteratively designed and evaluated as a grading tool through two live deployments during the Spring 2016 6.0001 staff exam grading sessions. Approximately two hundred students were enrolled in the course, and nine instructors used GroverCode to grade nearly all student code submissions. A total of seven programming exercises of a variety of difficulties were graded over the course of these two sessions. GroverCode was particularly appreciated on simpler problems where correct solutions were clustered together and graded by a single teacher's action. The canonicalization process applied to more complex solutions, especially those which included student-written objects, was at times more harmful than helpful for teacher's comprehension of student code. Conversely, the feature for grouping solutions based on their behavior on test cases was appreciated regardless of solution complexity.

Figures **??**, **??**, and **??** capture the GroverCode user interface during the second and final deployment of the Spring 2016 term. Unlike OverCode, there are both correct and incorrect solutions in view; they are differentiable by their different background colors, as well as a red "X" next to every failed test. To better explain each test failure, the solution's differing actual and expected outputs are also displayed.

The panel of progress indicators and filters is always in view. The rows of aligned sequences of green checks and red dashes represent "error vectors", the pass/fail results with respect to the ordered list of teacher-specified test cases. The checkbox next to each error vector allows the teacher to selectively view subsets of solutions based on the particular tests they pass and fail. Solutions with the same error vectors may include similar mistakes.

Correct solutions are stacked the same way they are in OverCode. In contrast, incorrect solutions were canonicalized but not stacked, even if they were identical after canonicalization. Grades are propagated to all the solutions in a stack; propagating actual student grades based on a probabilistic canonicalization process that was not thoroughly vetted before deployment would be unfair to students.

The user interface includes the following features in an attempt to minimize the cognitive load experienced by teachers rapidly switching between grading one solution and the next:

- Horizontally aligned solutions for side-by-side comparison
- Easy filtering of solutions by input-output behavior, i.e., their error vectors
- Canonicalized variable names
- Solutions ordered to minimize the distance between adjacent solutions with respect to a custom similarity metric defined in Terman **??**
- Highlighted lines of code in each solution which differ from the previous solution in the horizontal list

The hope is that canonicalization helps teachers switch between solutions more often than it harms reability.

The GroverCode user interface displaying solutions to an introductory Python programming exam problem, in which students are asked to implement a function to flatten a nested list of arbitrary depth.

A correct solution, its corresponding raw submission, and its performance on test cases, as displayed in GroverCode.

A stack with the rubric dropdown menu open. Text for each of the checked items is automatically inserted in the comment box.

The GroverCode implementation is a modification of the OverCode pipeline, including the updates described in Section **??**. Solutions which return an expected output for all test cases during the preprocessing step are categorized as "correct", and all other solutions, having failed at least one test case, are categorized as "incorrect". Correct solutions are canonicalized by the same process as was used in the original OverCode.

In the original OverCode pipeline for correct solutions, variable behavior is assumed to hold enough semantic information to be the sole basis on which variable names are canonicalized. In this first version of GroverCode, applies this rule to incorrect solutions as well, but only for variables whose behavior matches a common variable in the correct solutions.

However, that purely behavior-based renaming strategy may change in future versions of GroverCode because, given that incorrect solutions are known to be wrong with respect to input-output behavior, the behavior of the variables within them is suspect too, regardless of whether it happens to match a common variable in a correct solution.

Consider the following example: in the syntax of an incorrect solution, a variable i may be operated or depended on exactly the same way as a common variable in a correct solution. However, if an error somewhere else in the solution causes i to behave differently, the original method of variable renaming will be thrown off. Based on this example, variables in incorrect solutions that are not already renamed based on behavior are renamed based

on syntax.

The most recently updated OverCode pipeline represents each line's syntax and variables in a separable way (see Section **??**). The line's syntax, e.g., `for __ in __:`, is referred to as a template. For each common variable in correct solutions, GroverCode counts how many times it appears in each template and in which location, represented as an index into the blanks in the template. Examples of templates and locations are shown in Table **??**.

Example line of code Template Location `def power(base, exp):` `def power(__,__):` 1 `while index <= exp:` `while __<=__:` 1 `return 1.0*base*power(base, exp-1)` `return 1.0*__*__*power(__,__-1)` 3 `return base*power(base, exp-1)` `return __*power(__,__-1)` 2 `return power(base, exp-1)*base` `return power(__,__-1)*__` 1 `ans = base*power(base, exp-1)` `__=__*power(__,__-1)` 3 `if exp <= 0:` `if __<=0:` 0 `if exp == 0:` `if __==0:` 0 `if exp >= 1:` `if __ >= 1:` 0 `assert type(exp) is int and exp >= 0` `assert type(__) is int and __>=0` 0, 1

If a yet-unrenamed variable in an incorrect solution appears in the exact same template-locations as a common variable in a correct solution, it will be renamed to match that common variable.

If a variable in an incorrect solution is still not canoncialized, the counts of template-locations associated with each common variable in the correct solutions are used, as described in detail in Terman [**?**], to infer the most likely common variable it could be renamed to, as long as a threshold for similarity is met. Otherwise, its original name is kept.

The GroverCode analysis pipeline was run on both the midterm and final exam problems from the Spring 2016 semester of 6.0001, which had approximately 200 students enrolled. These exams contained seven programming problems in total, and between 133 and 189 solutions per problem made it through the analysis pipeline to be displayed in the user interface.

Using the GroverCode user interface, nine instructors, including one lecturer and eight

teaching assistants (TAs) graded these solutions as part of their official grading responsibilities. Those solutions that did not successfully pass through the pipeline were graded by hand afterwards.

The TAs' grading events, e.g., adding or applying rubric items and point valuesto solutions, were logged. An observer took extensive notes during each day-long grading session to capture spontaneous feature requests as well as bugs and complaints. For full disclosure, one of the TAs in these grading sessions was the Masters of Engineering student who implemented most of GroverCode, and the observer was the author of this thesis.

For each problem processed during the field deployments, an example of a staff-written correct solution is included below. Given the increasing difficulty of these exam problems, the following numbers dropped between the midterm and the final: the number of students present to take the exam, the number of students who submitted any solution to a given problem before the test period ended, and the number of correct solutions that could be clustered.

Midterm Problems

Question 4: `power`. Write a recursive function to calculate the exponential `base` to the power `exp`In the future, new user interfaces would be helpful for this task, especially one which helps humans compare the output of models with different parameter values.

Question 5: `give_and_take`. Given a dictionary `d` and a list `L`, return a new dictionary that contains the keys of `d`. Map each key to its value in `d` plus one if the key is contained in `L`, and its value in d minus one if the key is not contained in `L`.

Question 6: `closest_power`. Given an integer base and a target integer `num`, find the integer exponent that minimizes the difference between `num` and `base` to the power of exponent, choosing the smaller exponent in the case of a tie.

Final Exam Problems

- Question 4: `deep_reverse`. Write a function that takes a list of lists of integers `L`,

and reverses `L` and each element of `L` in place.

- Question 5: `applyF_filterG`. Write a function that takes three arguments: a list of integers `L`, a function `f` that takes an integer and returns an integer, and a function `g` that takes an integer and returns a boolean. Remove elements from `L` such that for each remaining element `i`, `f(g(i))` returns `True`. Return the largest element of the mutated list, or -1 if the list is empty after mutation.
- Question 6: `MITCampus`. Given the definitions of two classes: `Location`, which represents a two-dimensional coordinate point, and `Campus`, which represents a college campus centered at a particular `Location`, fill in several methods in the `MITCampus` class, a subclass of `Campus` that represents a college campus with tents at various `Locations`.
- Question 7: `longest_run`. Write a function that takes a list of integers `L`, finds the longest run of either monotonically increasing or monotonically decreasing integers in `L`, and returns the sum of this run.

The number of submissions submitted for each problem and the number that were successfully processed by the GroverCode pipeline is shown in Table **??**.

q4 q5 q6 q4 q5 q6 q7 Submissions 193 193 193 175 173 170 165 Mean lines per submission 9.9 16.9 19.8 12.3 20.9 50.0 41.8 Solutions successfully processed 186 189 168 170 166 134 133 (96%) (98%) (87%) (97%) (96%) (79%) (81%) Number of submissions submitted and successfully processed by GroverCode for each problem in the dataset. Reasons why a solution might not make it through the pipeline include syntax errors and memory management issues caused by students' inappropriate function calls.

Table **??** captures the scale of the variation as well as some clustering statistics.

q4 q5 q6 q4 q5 q6 q7 Correct submissions 182 160 94 96 49 16 12 (94%) (82%) (49%) (55%) (28%) (9%) (7%) Incorrect submissions 4 29 74 74 117 118 121 Test cases 10 15 25 11 10 17 28 Distinct error signatures 6 16 36 12 38 57 42 Correct stacks 40 84 93 47 46 16 12 Stacks containing > 1 submission 13 18 1 8 2 0 0 Solutions collapsed into stacks 151 94 2 57 5 0 0 The degree of variation in input-output behavior and statistics about stack sizes.

Table ?? summarizes the counts of the various mechanims by which variables in incorrect solutions were canonicalized.

q4 q5 q6 q4 q5 q6 q7 Vars. in incorrect submissions 15 149 482 289 550 559 859 Vars. renamed based on values 14 84 266 97 246 97 187 Vars. renamed based on templates 0 58 166 136 264 188 489 Vars. not renamed 1 7 50 56 40 274 183 Statistics about variables renaming based on different heuristics in the GroverCode canonicalization process.

For simpler solutions, variable renaming was an invisible and possibly slightly confusing helping hand. One grader remarked, outloud: "Why is everyone naming their iterator variable 'i'?" at which point he had to be reminded of the variable renaming process. As solutions became more varied and structurally complex, graders started immediately looking the raw solutions because the renaming of variables, removal of comments, and standardization of whitespace in the canonicalized solutions was removing clues they needed in order to understand the student's intent.

While it was difficult to get direct feedback on the helpfulness of pair-wise difference highlighting and optimized solution ordering, graders heavily used and appreciated the ability to filter and grade solutions one error vector at a time. At least one grader remarked aloud that it seemed like many of the solution with the same error vector made similar mistakes. Therefore filtering by error vector may have been one of the stronger contributors to any hypothetical decreased cognitive load due to using GroverCode over the status quo of random assignment to solutions in a CSV file.

1.4 Conclusion

The clustering described in the original OverCode work was relatively limited in scope, but it did produce , at least for simple introductory Python programming problems, a concise standardized representation of platonic solutions that can be used for more statistically sophisticated clustering techniques and as a starting point for helping graders understand

and grade incorrect student solutions by hand.