ACES: Automatic Evaluation of Coding Style

Stephanie Rogers
Dan Garcia
John F. Canny
Steven Tang
Daniel Kang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-77
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-77.html

May 15, 2014

Acknowledgement

I would like to thank my advisors, John Canny and Dan Garcia, for their help through out my entire research project and encouragement in applying to various conferences. Special thanks go out to Steven Tang and Daniel Kang, for their invaluable help and contributions to this project.

I would also like to thank Siebel Scholars Foundation, Google, and ACSA for funding my Master's year at Berkeley. And finally, thank you to all of my friends who helped me through the process in various ways, especially James Huang who helped me when possible and supported me always.

ACES: Automatic Coding Evaluation of Style

by Stephanie Rogers

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:

Senior Lecturer SOE Dan Garcia
Research Advisor

(Date)

* * * * * * *

Professor John Canny
Research Advisor (Second Reader)

(Date)

ACES: Automated Coding Evaluation of Style

by

Stephanie Rogers

B.A. UC, Berkeley 2013

A thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science
in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Senior Lecturer SOE Dan Garcia, Chair
Professor John Canny

Spring 2014

ACES: Automated Coding Evaluation of Style

by

Stephanie Rogers

Abstract

ACES: Automated Coding Evaluation of Style

by

Stephanie Rogers

Master of Science in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Senior Lecturer SOE Dan Garcia, Chair

Coding style is important to teach to beginning programmers, so that bad habits don't become permanent. This is often done manually at the University level because automated static analyzers cannot accurately grade based on a given rubric. However, even manual analysis of coding style encounters problems, as we have seen quite a bit of inconsistency among our graders. We introduce ACES–Automated Coding Evaluation of Style–a module that automates grading for the composition of Python programs. ACES, given certain constraints, assesses the composition of a program through static analysis, conversion from code to an Abstract Syntax Tree, and clustering (unsupervised learning), helping streamline the subjective process of grading based on style and identifying common mistakes. Further, we create visual representations of the clusters to allow readers and students understand where a submission falls, and what are the overall trends. We have applied this tool to CS61A–a CS1 level course at UC, Berkeley experiencing rapid growth in student enrollment–in an attempt to help expedite the involved process of grading code based off of composition, as well as reduce human grader inconsistencies.

Contents

List of Figures

Acknowledgements

I would like to thank my advisors, John Canny and Dan Garcia, for their help through out my entire research project and encouragement in applying to various conferences. Special thanks go out to Steven Tang and Daniel Kang, for their invaluable help and contributions to this project.

I would also like to thank Siebel Scholars Foundation, Google, and ACSA for funding my Master's year at Berkeley. And finally, thank you to all of my friends who helped me through the process in various ways, especially James Huang who helped me when possible and supported me always.

Chapter 1

Introduction

Code is read much more often than it is written. Computer programs should be written not only to satisfy the compiler or personal programming "style", but also for "readability" by humans. Coding with good style helps programmers identify and avoid many errors, and results in code that is more readable, secure, extensible, and modular.

Thus, programming style has started to become more formalized, with a set of rules and guidelines. Coding conventions have become the norm: most often designed for a specific programming language. PEP8 is the style standard for the programming language of Python, which comes from Python Enhancement Proposals (PEP). Static code analysis tools (e.g., lint checkers) attempt to enforce these strict Python style standards on source code. In practice, most style enforcement actually comes in the form of code reviews: manual analysis by experienced human readers. Furthermore, style analysis is often done manually in classes that teach Python because automated Python static analyzers cannot accurately grade based on a given rubric. However, even manual code style evaluation has problems, as we have seen quite a bit of inconsistency from our CS1 graders due to the subjective nature of grading code based on composition.

We want students at the University level to practice good techniques while coding, to prepare them for industry or life beyond academia. Nontrivial machine grading of student assessments is an emerging problem, as Massive Open Online Courses (MOOC) become more popular. Aimed at unlimited participation, these classes–as well as high enrollment university classes–face the problem of scalability with respect to grading. Automating the manual processes of grading becomes a highly-relevant and arguably necessary approach to expanding the ability of these classes to evaluate both the learning outcomes and the quality of the assessment.

We introduce the Automated Coding Evaluation of Style (ACES), a system that attempts to help automate the process of code reviews by predicting a score for the composition of computer programs. ACES assesses the composition of a program through static analysis, feature extraction, supervised learning, and clustering, all of which help to semi-automate the subjective process of grading based on style. It also helps identify common mistakes. The purpose of the tool is two-fold: to provide highly detailed and targeted feedback based off of the features that mattered while grading the code, and to act as a verification tool, to enforce consistency between graders and for any particular grader. While completely automated systems are ideal, our attempts to fully automate the process proves to be a challenge. Out of a desire to keep the human element while grading code based off of style in order to ensure correctness, we apply our findings as features in a code review tool which helps streamline the process. Moreover, we are able to extrapolate grading results from our tools, including what common mistakes students make and what features are considered more important.

Chapter 2

Related Work

2.1 Coding Composition Evaluation

A few ways in which code is currently automatically graded include unit tests, static analysis checkers, and code coverage analyzers. The most widely used approach to automating the grading of student code is through test-cases. In Fully Automatic Assessment of Programming Exercises, Saikkonen et al. checked student exercises by allowing the instructor to specify test cases. While this tool primarily focused on the correctness of the student's code, it also allowed for analysis on the program's structure and other related factors, which correspond to the coding style [20].

Another common practice, especially in industry, is through a static analysis checker. Lint (software) refers to syntactic discrepancies in general, and modern lint checkers are often used to find code that doesn't correspond to certain style guidelines. Specifically, there are several static analysis style checkers in Python that enforce Python code style and standards. PyLint is a static code analysis tool for enforcing Python style on source code [18]. It can highlight when the code contains an undefined variable, when code is imported but not used, as well as highlighting other bad techniques. It can be a bit verbose, complaining about things like lines being over 80 characters long, variables not matching a specific regex, classes having too few public methods, and methods missing docstrings.

PyLint also generates a "code report", including how many lines of code, comments, docstrings, and whitespace the file has, the number of messages per-category, and gives code a "score" from 0 (syntax error) to 10 (no messages). Other linters like PyFlakes [17], and PEP8 [14] do very similar things without the scoring.

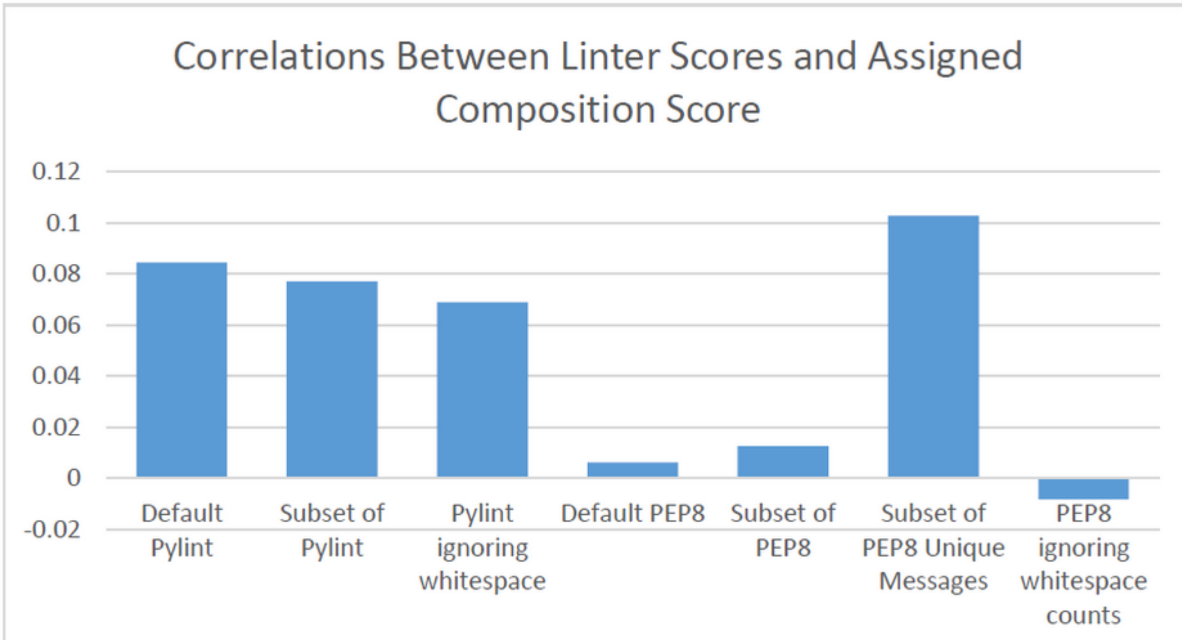

Figure 2.1. Lint Software Correlation

Some of the features extracted by these lint tools can be used in our own analysis. However, we noticed two important things. The scores or numbers of suggestions produced by all three of the above lint tools (including subsets) were not correlated with the assigned composition scores for our CS61A project, as shown in Figure 2.1. Additionally, we have noticed that several important features were not taken into consideration. Our tool differs from these lint checkers, and is unlike most of these lint tools, in that we actually attempt to learn from past submissions in order to understand what factors of the coding style matter. Grading style through static analyzers can be quite limiting, and is not always helpful in actual grading. For this reason, there are several tools which use a more feature-based or dynamic approach to grading code based on style.

A common goal of the related work cited below is simply to allow teachers to oversee their classes or provide solution based feedback to students. In terms of the more open-ended grading of code based on structure or composition, nothing has dominated the field. Aggarwal et al. extracted features from code by converting it to a data dependency graph and annotating that graph with control structures [1]. They then used common supervised learning algorithms to automate the scoring process. Several problems arise from this approach, as the features seem arbitrarily constructed and encounter a powerset problem. Taherkhani et al. identified which sorting algorithm a student implemented using supervised machine learning methods [23]. Each solution was represented by statistics about language constructs, measures of complexity, and detected roles of variables. Luxton-Reilly et al. labeled types of variations as structural, syntactic, or presentation-related [11] . The structural similarity was captured by the control flow graph of the student solutions. Singh et al. took a slightly different approach by allowing the instructors to supply an extremely formal model of possible errors, in essence creating a program based on constraint satisfaction [22]. They found the smallest set of modifications that would fill in the program holes to determine correctness. While this solution is unique, in that it can point out complete sets of errors (i.e. more than one), it is highly constrained to those problems which only have a small set of common errors. In other words, the specification must be complete and a black box in order for this solution to succeed.

Although not specifically related to code grading or composition, there is quite a bit of research that attempts to grade open-ended questions or short answer responses including peer or self-grading [15], [19], [24], feedback [13], [15], and clustering [2], [4]. Brooks et al. used clustering to grade open-ended short answer questions quickly and at scale [4]. They proposed a cluster-based interface that allowed teachers to read, grade, and provide feedback on large groups of answers at once, similar to what we prototyped for ACES. Additionally, they compared this interface against an unclustered baseline, and found that the clustered interface allowed teachers to grade substantially faster, to give more feedback to students, and to develop a high-level view of students' understanding and misconceptions. We performed a similar, but smaller, user study with our prototype to show that not only does it make grading more efficient, but it also has the potential to make the process more consistent and accurate.

Glassmen et al. [8] also used a clustering approach, but with a focus on feature-engineering and specifically for the grading of code based off of composition. They performed a two-level hierarchical clustering of student solutions: first they partitioned them based on the choice of algorithm, and then partitioned solutions implementing the same algorithm based on low-level implementation details. They found that in order to cluster submissions effectively, both abstract and concrete features needed to be extracted and clustered.

In previous studies, researchers have used program similarity metrics to identify plagiarism (such as MOSS and SIM) [7], [21] and modeling student solutions [16]. Our work is primarily based off of Huang et al., who examined the syntactic and functional variability of a huge code base worth of submissions for a MOOC class using structural similarity scores [10]. Their strategy was to convert the C++ code submissions to Abstract Syntax Trees, in the form of a language-neutral Javascript Object Notation (JSON) data structure, and then clustered and visualized the submissions that contained similar structure. We applied this technique to our own set of data, analyzed the results to confirm their original hypotheses, and built a tool to automate the entire process based off of their assumption of clustering as a viable technique to grading.

2.2 Autograder Tool

Web-based feedback mechanisms have become the ideal way to provide feedback to students at mass scale. Our tool incorporates many successful features of prior systems, including inline comments [9], summary comments [12] and comment memory [5]. DeNero et al. created an online tool that allowed instructors to perform code reviews and provided feedback efficiently at scale [5]. It adapted Google's open source code review system, but mainly provided the extra functionality of comment memory, an idea very similar to the one explored in our research.

Studies of similar web-based feedback systems have shown obvious improvements in staff and student user experience. MacWilliam et al. [12] and Bridge et al. [3] both found that the total time taken to provide feedback was substantially reduced using a web-based system versus a PDF-based process. From the student perspective, Bridge and Appleyard reported that students prefered to submit assignments online and Heaney [9] found that online feedback also correlated with higher exam scores [3].

Chapter 3

Supervised Learning Approach

When we began our project, we were attempting to automatically predict the score of a given submission by using past submissions and their relative composition scores through supervised learning. Our attempts were applied to UC Berkeley's introductory course in computer science, CS61A, for one particular project on Twitter Trends in the Python programming language. This project has heavy skeleton code and has stayed consistent for 4 semesters, offering us the unique opportunity to analyze 1500 student code submissions. All 1500 submissions had been manually scored and commented on by human graders for the course based on coding composition. Students' coding submissions were given a score of {0, 1, 2, or 3}, with 3 given to code with no style errors, and subsequently decreasing with more composition mistakes.

These labeled submissions scream for a supervised learning approach. Using meaningful features of coding style, we can learn from these past submissions and their relative grades to predict scores for those coding submissions in the future which have not been graded by human graders. To maintain consistency, our output would also be one of {1, 2, or 3}, excluding 0 as those were given to no submission or no code. Accuracy would be measured by comparing our predicted scores to the actual labels, or past grades. As we used several approaches that predicted a fractional number rather than a categorical class, we defined this to be within 1 integer score from the actual received grade.

All Features Extracted	
Repeated Code	Repeated Function Calls
Length of variable name	Length of function names
Average, Max, Min Length of line	Line too long
Average Length of code (per function)	Length of entire code
Meaning of variable name	Contains capital O or lowercase l
Commented out code	Docstring only at beginning of function
Code coverage	Unreachable Code
Number of key words	Redundant if statements
Unnecessary semicolons	is and is not instead of == and !=
Pylint score	More than one statement on a single line
Unused variables	Assigning to function call with no or None return
Redefining built-in	Too many returns
Code Complexity	Number of Exceptions

Table 3.1. List of Features

3.1 Feature Extraction

A Python script, which extracts features of the code through dynamic and static analysis, and produces a feature vector per submission, was run across all submissions. In order to determine what features were important while grading composition in the past, 15 veteran graders filled out a questionnaire about important stylistic features, the head grader for this semester was interviewed, and the rubric for composition scoring for this particular project was carefully examined. From this research, over 35 significant features were extracted. We filtered down to 26 features based on what we could implement and what we believed to be important. All 26 features are listed in Table 3.1. Appendix A shows examples of buggy code for 10 of these features.

3.2 Learning

At this point, several different learning algorithms were incorporated to train our classifier, *mostly* using the *Sci-kit Learn* Python library, which has built-in implementations for several machine learning techniques. The main approaches taken were multinomial logistic regression and linear regression. From there, 10-fold cross-validation–training on 9/10 of the dataset and testing on the remaining 1/10 for all 10 folds–was run on approximately

600 submissions. However, even after optimizing and twiddling with the amount of regularization, we were only able to reach an accuracy of prediction of 53%. As there were three classes to predict, accurately predicting only 53% of the scores is better than random, but not nearly as high as expected. We found that in most cases, the accuracy was a result of choosing the class with the highest frequency.

3.2.1 Improvements

Our first thought was to apply different learning methods. However, even after trying different learning methods such as Random Forests and SVM, there was no obvious improvement. With random forests we were able to look at which features were being chosen for the decision trees. This allowed us to determine which features mattered more, or were more discriminable. At each level in the decision trees, the feature that reduced entropy the most through a binary split was chosen, making it the most discriminable feature at each level in the tree.

As we found that most of the misclassification came from scores of 3 being misclassified as a 2, we decided to delve deeper into the boundaries. We trained binary classifiers at each boundary. The code and it's relative feature vector would be classified as a 1 or a 2, and if it was a 2 from this classifier, we would subsequently classify it as either a 2 or a 3 using the second binary classifier. The 1-2 classifier had an accuracy of 80%. However, this was due to the titled distribution towards 2s. The 2-3 classifier had an accuracy of 60%, making this boundary harder to distinguish. Even human graders admitted in our questionnaires that the distinction between a 2 and a 3 seemed subjective and, at times, arbitrary.

Our learning showed extremely high bias: we found both that the training error and cross-validation error were high. We were significantly under fitting our training data, performing equally bad on the training, cross-validation, and testing error. In this case, the model does not fit the training data very well, and it doesn't generalize either. Usually high bias means that there were not *enough* features, but for this project, there simply were not enough features that were *meaningful*. As feature selection was a potential problem, we attempted to reduce our feature dimensionality using Principal Component Analysis and Linear Discriminant Analysis, to convert our features before attempting to learn. This was done on the original set of 26 features. We also iterated through various dimensions, attempting to see which was best. LDA increased our accuracy all the way to 58%, as it used the labels to find the most discriminatory features between classes.

We then redid feature selection altogether, and attempted to use more specific features of this particular project for CS61A, trimming our number of features all the way down to the 10 listed in Appendix A. This reached 60% accuracy, as these were features of higher discriminability, which were manually found using exploratory data analysis, decision tree feature information, and even the LDA results. If given more time, feature selection would have been done even more carefully, as it had the most potential in improving our results.

3.3 Discussion

3.3.1 Features

After plotting each function's distributions using box plots, we manually analyzed the visualizations. While the average of each feature tended to be correlated with the corresponding composition score, the variance for each feature was simply producing way too much noise for us to overcome. In most cases, the overlap between features was so high, that even regularizing to account for the noise didn't help. Two example feature box plots are shown in Figure 3.1. The *average function length* decreases as the composition score increases, showing that more highly graded code is more concise. Similarly, the *average number of duplicate lines*, which was computed using Pylint's "repeated code" feature, decreases with better composition scores. However, in both box plots we see extremely high overlap in the distributions. Once features with higher discriminability were chosen, we were able to increase the accuracy. Flattening the features through the use of square root and log of each feature value produced no noticeable difference in the accuracy of our model, despite being one way to overcome the distribution issue.

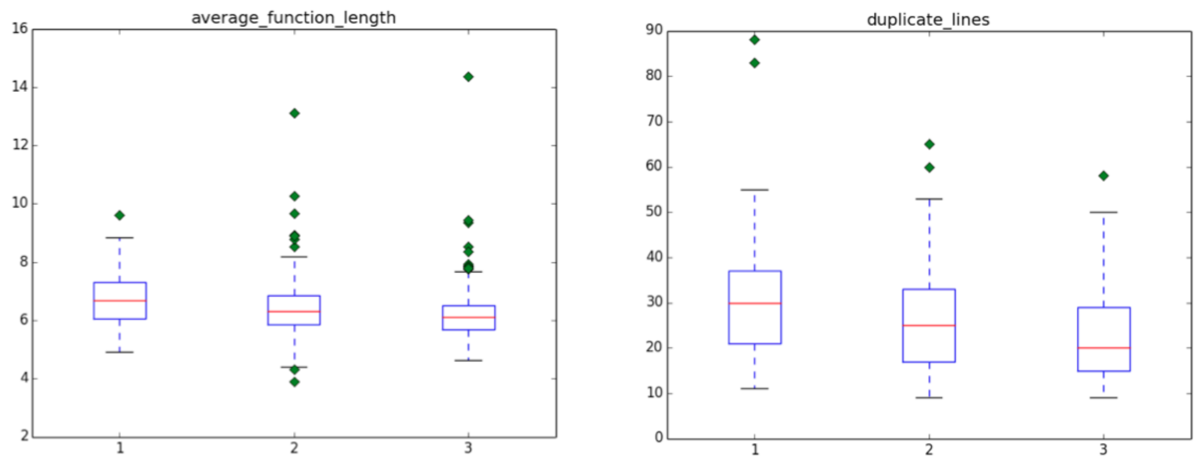

Figure 3.1. Boxplots: Function Length and Duplicate Lines per Composition Grade

3.3.2 Inter-rater Reliability

The main explanation for these results, comes from the fact that humans can not consistently grade based off of composition. During our research, we performed two separate experiments to test inter-grader reliability, i.e. how consistent were graders at grading code based off of composition. In the first experiment, we had graders from the CS61A Fall 2013 semester grade random coding submissions from Fall 2012 (which already had been assigned a particular grade). We found that only 55% of the submissions were given the same grade. As the grade was only a score from 1-3, this result was a bit worrisome. We noticed that graders for Fall 2013 tended to grade a bit harsher, so we decided to pursue the same experiment within the same semester: having current graders grade other submissions for Fall 2013. Here, we found that only 47.5% of the submissions were given the same grade! We conclude that human graders are extremely inconsistent while grading code based off of composition. As our implementation of supervised learning attempted to learn from human grades, it seemed that its success was limited to the consistency of that of a human. These results make us wonder what exactly it means to have humans grade assignments as opposed to computers. Perhaps this shows that computers should be doing all of the work, as they can be more consistent.

3.4 Future Work

As we only tested inter-grader reliability, it would be interesting to test intra-grader reliability as well. This would tell us how consistent graders are themselves. Does the grader get tougher as they see more code and more errors, or easier as they get tired of grading? Or are they extremely consistent in the grades they are giving? If shown to be consistent, then it would make sense to normalize the scores–or labels–by the grader's average score. Since each grader has a different distribution, adjusting scores per grader should improve annotator agreement and thus accuracy. Our proposed normalization of scores would be the following:

$$\frac{overall\_mean}{grader\_mean} * given\_score = normalized\_score$$

As we could not learn from an inconsistent source, it made sense to progress to a more unsupervised approach where we did not depend on the inconsistencies of humans. Thus, our next approach was to use clustering to find common mistakes and potentially identify code samples that are similar enough to receive similar scores.

Chapter 4

Clustering Approach

This section discusses our second approach to auto-grading style by identifying common approaches that students took when defining their functions in the class project. The motivation behind this second approach is that there might only exist a limited number of common approaches that students take when solving a function, and if it's possible to identify which common approach a student used, then feedback or grades can potentially be automatically generated. To identify different approaches, we computed the edit-distance between one project submission and all other project submissions (using the abstract syntax tree of the submission), and repeated this computation for each submission. Then, submissions with low edit-distances were clustered together, and these clusters were then manually inspected to identify the style approach that was taken within that cluster. The first section describes how we implemented this clustering process, and the following sections describe results and analyses from this process.

4.1 Implementation

First, the decision was made to look for common structural approaches for individual functions, rather than for entire project submissions. This was motivated by the fact that most functions in the project could be structured completely independently from one another, so it made sense to look for common stylistic approaches per function rather than per project. Each function submission was converted into its abstract syntax tree (AST) represented as a language-neutral JavaScript Object Notation (JSON) object–a lightweight data-interchange format–with a very specific format. This JSON object anonymizes most variables and function names, unless otherwise specified by the user, as names are typically unimportant when considering structure. However, variable and function names that occurred as part of the project skeleton code were not anonymized, as it is structurally significant when students use variables or functions whose values and effects were consistent between projects. Naming is obviously an important part of coding composition, so this represents one major flaw with our approach.

In order to determine how structurally similar two submissions were, we used the tool from J. Huang et. al. [10], which when given two JSONs, could calculate the *AST Edit Distance* between two ASTs. This pairwise computation used a dynamic programming algorithm, which considered the trees unordered. Thus, pairs of submissions with low edit distances could conceivably be similarly structured, as it would not take many edits to reach an equivalent AST. This pairwise process is quartic in time with respect to the AST size, but only needs to be run once on a particular set of submissions, acting as a one-time cost for preprocessing. The process could likely benefit from optimizations.

Lower edit distances meant that submissions were more similar. Thus, the inverse of this edit distance was used as our *similarity score* between submissions. With similarity scores calculated pairwise for all submissions, visualizations of common structural approaches to functions could be created. The program Gephi is used, which is an interactive visualization and exploration platform for networks, complex systems, and graphs [6].

4.2 Results

In Figure 4.1, we can see the final visualization produced by Gephi. Each node in the graph corresponds to an AST of an original function submission. Edges where the similarity score between two functions was below a certain threshold were not included in

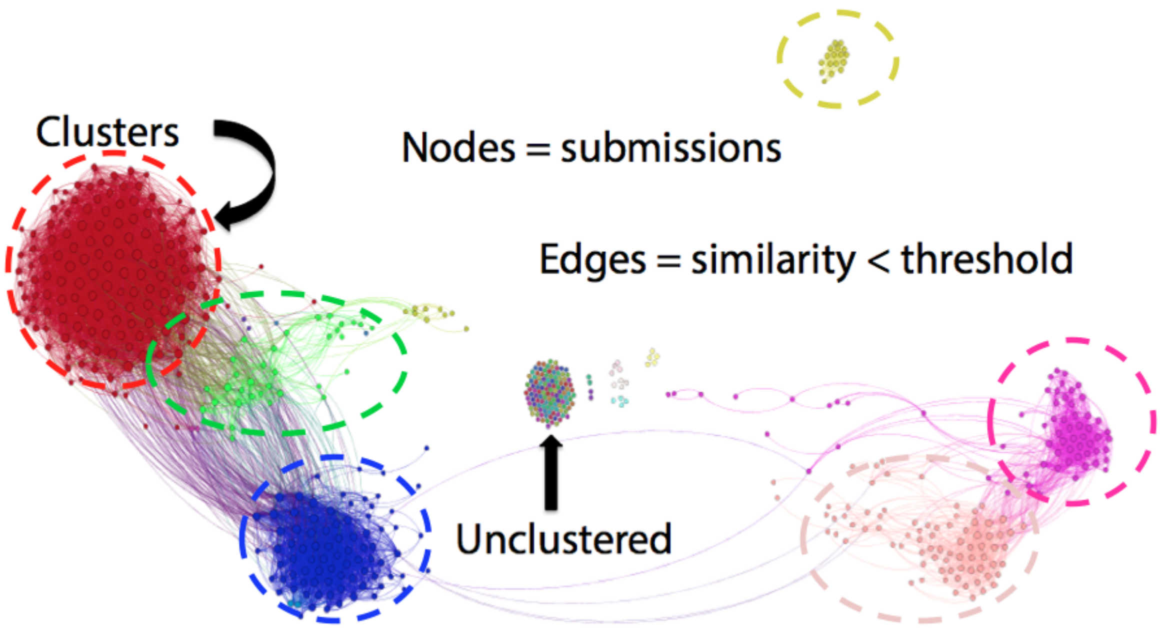

Figure 4.1. Clusters with Annotations: Fall 2013 data, Threshold = 5%

the graph, since low similarity scores means the two submissions should not be considered similar. Thus, an edge can be thought of as connecting two nodes that have a high enough similarity score to warrant analysis. For example, with a threshold of 5%, only the highest 5% of edges are kept, thus indicating that edges are likely to have meaning. By examining the network of solutions only in the top 5%, we observe a smaller, more manageable number of common solutions or mistakes–only 6 main clusters appear. By organizing the space of solutions via this network, clustered nodes are syntactically similar.

In order to produce the cluster visualization as shown in Figure 4.1, Gephi runs a ForceAtlas algorithm, which uses repulsion and gravity to continuously push and pull nodes away from each other. Nodes that have a high similarity score between one another are attracted to each other, and nodes without a high enough similarity score repel from each other. The distance between nodes and clusters directly correspond to the syntactic similarity: nodes and clusters closer to each other share similar structure.

The clusters are colored by modularity, a measure of how well a network decomposes into modular communities. The group of multi-colored nodes represents those submissions that were unclustered and therefore highly dissimilar in structure to most other submissions:

somewhere around 20-25% of the submissions remained unclustered depending on the input submissions and threshold. These particular submissions would have to be individually, manually analyzed and would not be a part of the automated process due to their unique style of solution.

4.3 Analysis

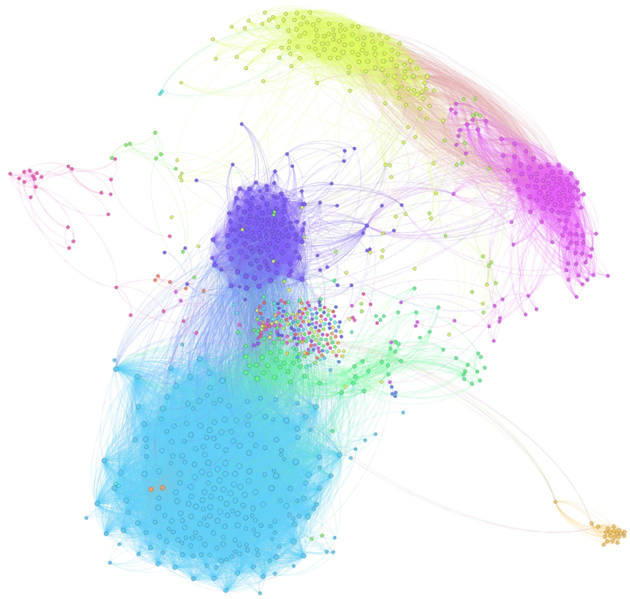

Figure 4.2. Clusters: Fall 2012 and Fall 2013 data, Threshold = 5%

4.3.1 Multiple Semesters

We ran the clustering visualization with different subsets of the 4 semesters worth of data that we had. Even with additional semesters-worth of data, we found that the clusters did not change significantly. We show the progression of the clusters as more semesters worth of submissions are added to the visualization in Figure 4.1, which represents just Fall 2013 submissions, Figure 4.2, which represents just Fall 2013 and Fall 2012 submissions, and Figure 4.3, which represents all submissions from Fall 2012, Summer 2012, Spring 2013, and Fall 2013. The number of common structural solutions, or the number of clusters in

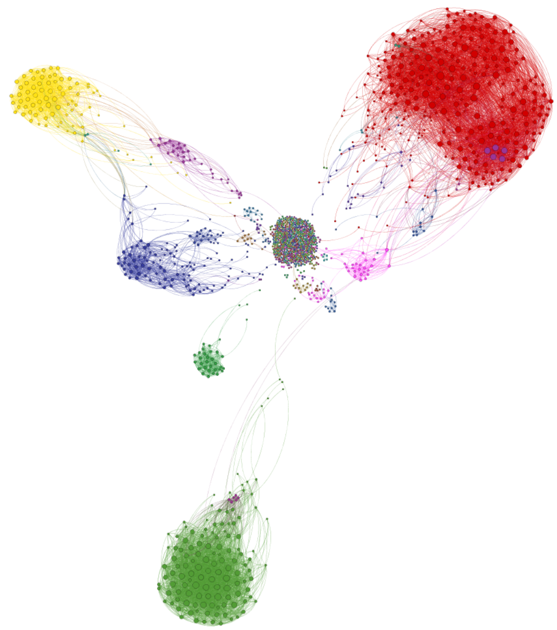

Figure 4.3. Clusters: All 4 semesters, Threshold = 2%

the visualization, for the find_centroid problem, remained at 6 even when 1000 additional submissions were added to the visualization, as shown in Figure 4.3. The figures show that the same clusters remain, however, we do see the clusters expanding away from each other, as we get additional submissions.

It is important to note, that since there were so many more nodes, we decided to set our threshold to view only the top 2% of edges when considering all 4 semesters, thus removing thousands of edges, but not affecting the clusters in any significant way. With these conclusions in mind, we decided to focus on the dataset from Fall 2013 alone for analysis, as it would be more manageable.

4.3.2 Manual Classification

We proceeded to do an in depth manual analysis on the clusters. For 3 different functions within the project 2 submissions–`find_centroid`, `find_center`, and `analyze_sentiment`–we performed manual analysis on the visualizations produced. In Figure 4.4 we can see one result of this manual classification. Here, from each of the 6 clusters produced, we

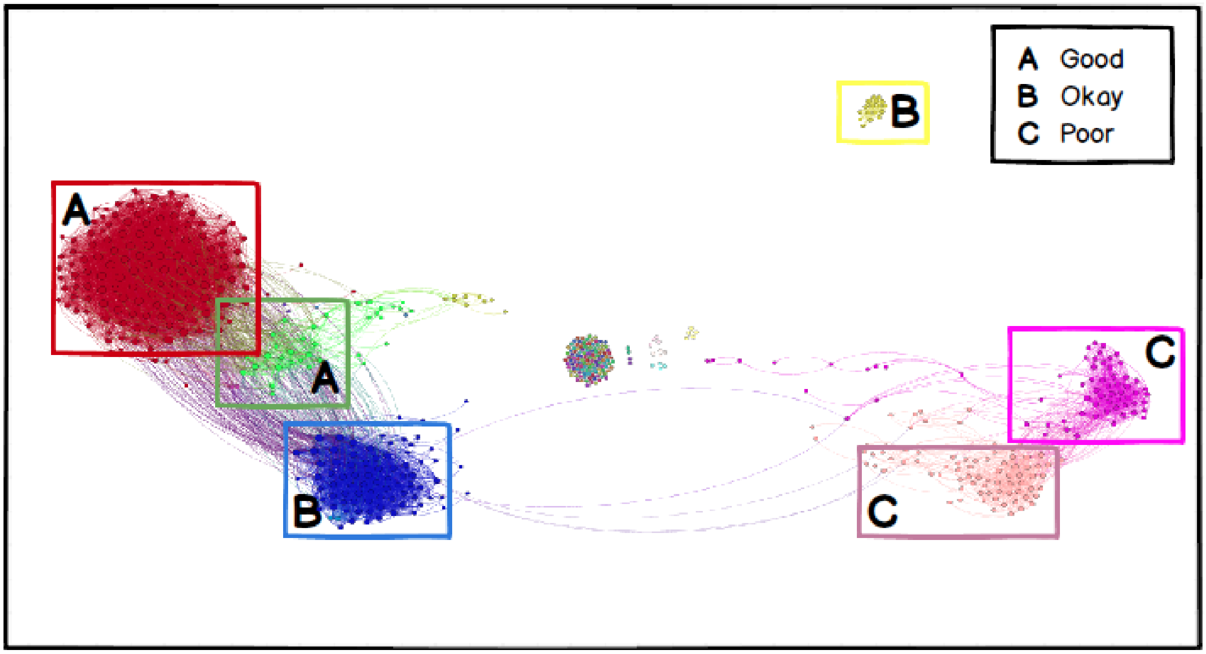

Figure 4.4. Manual Classification of Clusters

blindly and randomly selected 10-15 submissions. We sifted through the code for the `find_centroid`, labeling each as "Good," "Okay," or "Poor" style, based off of our own understanding of Python composition and the function itself. This is similar to the 3 categories of grades that were awarded to students, again excluding 0 as these were only for no code submissions. We were unaware of what cluster each submission belonged to at the time of labeling; we were blindly classifying. Figure 4.4 shows what label was eventually assigned to each cluster, by simply taking the majority for each submission analyzed in that cluster. In each case, it was clear that the submissions contained extremely similar style and structure. We realize that there could be strong bias in this manual classification, as we, the researchers, were the ones to perform these manual operations.

4.3.3 Clustered by Score

In order to provide some validation to the above claims and Figure 4.4, we decided to color the nodes by the original composition scores given to each submission, as we can gain some insight from the visualization about composition of each cluster. In Figure 4.5, in

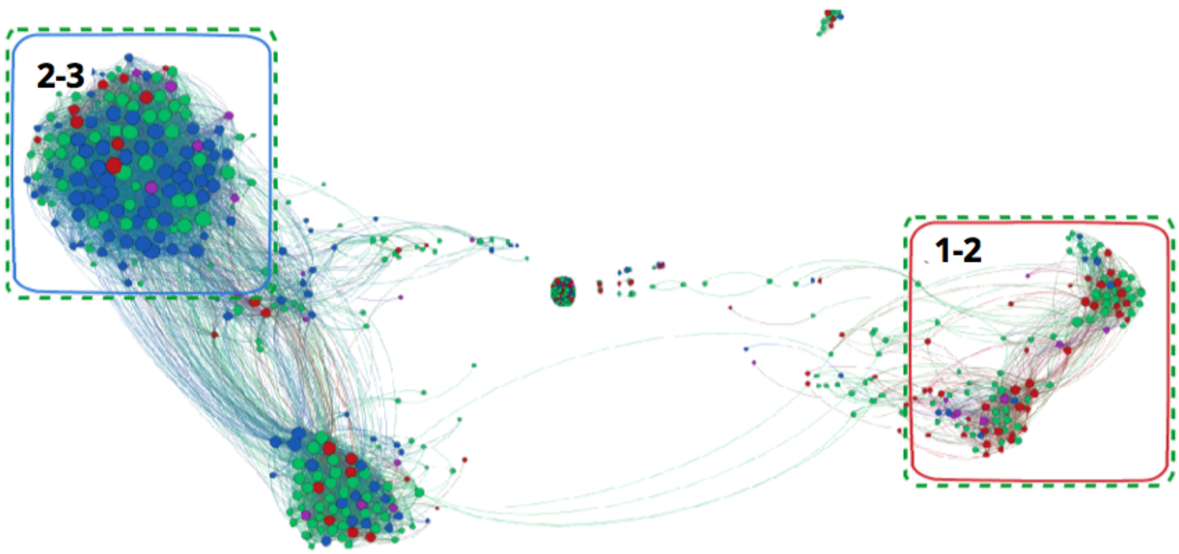

Figure 4.5. Clusters Colored by Original Composition Score

the upper left cluster, we can see that the submissions tended to be in the 2-3 range of composition scores, with clearly more 3s than any other cluster. Additionally, most scores in the two right clusters fell in the 1-2 range. This shows a general trend of each cluster. There are two major explanations for the lack of total confidence and variation of colors, even when there are obvious trends. The first is that this score is given to the entire submission as a whole, instead of per function. We are clustering on a per function basis, while the composition score was assigned to the entire submission. This explains why some red nodes end up in the higher 2-3 cluster. After manually analyzing those nodes that showed up red in the upper left cluster, we could see that the composition score was assigned due to style errors in other functions of the project submissions. As for the particular function clustered here, the composition was nearly perfect. While the entire submission was given a lower score of 1, this particular function had perfectly fine, 3-quality composition. We manually confirmed this for all the red nodes in the upper left cluster for this particular function. The second reason for this variability comes from the fact that readers simply are not consistent, as shown in the *Supervised Learning* chapter. Readers may have easily graded one submission of the same structure completely differently than another, often erring by 1 entire score.

A major conclusion from this analysis is that readers should potentially grade CS61A projects by problem (function) rather than as a whole. Grading composition at the function level can be meaningful and even partially automated. Limiting the scope of code that a reader has to rate might additionally improve reliability in a significant way, especially if a function is only graded by 1 or 2 readers total. Finally, the professor could formally specify the weight that each function has in the final composition grade, while readers usually have to determine this individually (another source of variability). Often readers will assign importance to style errors in incorrect ways, marking a student down an entire point simply for missing some white space, while others consider this trivial and may allow students to make several errors before considering a point deduction.

4.3.4 Common Mistakes

Finally, we manually analyzed the "Poor" quality clusters on the right to find common mistakes within the clusters. We found that these mistakes, including "under abstraction," "excessive while loops," "line length," and "repeated expensive function calls," occurred in over 80% of the submissions that we randomly sampled within those clusters. These common mistakes were extremely specific, as they were found per function call, and could thus be easily applied as potential feedback for the entire cluster.

BAD

```
for x in polygons:
    lat += find_cent(x)[0]*find_cent(x)[2]
    lon += find_cent(x)[1]*find_cent(x)[2]
    a+=find_cent(x)[2]
```

Underabstraction
Repeated function calls
No multiple assignment

GOOD

```
for p in polygons:
    clat, clon, pol_a = find_cent(p)
    lat += cen_lat * pol_a
    lon += cen_lon * pol_a
    a += pol_a
```

Figure 4.6. Common Solutions for `find_centroid`

Specific examples of common mistakes and common solutions were found in our analysis of the `find_center` function as shown in Figure 4.6. In the upper right cluster labeled C in Figure 4.4, we found that over 93% of solutions failed to save the results of their time-intensive function calls, resulting in inefficiency and repeated code. In Figure 4.6, we see an example of the bad solution that made the mistake of calling `find_centroid`, an expensive function, 5 times. More than 80% of solutions in that same cluster also made mistakes such as under abstraction and failing to do multiple assignment, although the latter is not necessarily a style error but a personal preference.

Chapter 5

Automation

5.1 Visualization Tool

The previous analyses were done on existing submissions from past semesters. If the project is reused in future semesters, then the work done in identifying and assigning meaning to clusters can be applied to these submissions in the same way it was done in this research. For example, if a new submission clusters with an already identified cluster, then that submission can be given the same feedback or score that was already established.

Identifying the need to automate the process described in the previous sections, we created an automated tool that can accept a new code submission as a Python file to automatically produce a visualization with the existing clusters that were already generated. As shown in Figure 5.1, the new code submission is emphasized in size and in color, so that the user can identify where the new submission is located. In this way, a grader can easily run the tool with a new submission and visually identify where the new submission belongs. Additionally, textual output of which cluster id the submission belongs to is output according to the modularity group the submission is clustered into.

The tool consists of two steps. The first is to run the Python script with this new submission, which converts it to the AST and adds a new row and column to the pairwise edit distance matrix. The second step is to run a Java script, which uses the Gephi toolkit to

Figure 5.1. Output of Visualization Tool with Highlighted Node in Black

automatically run all of the above processes, and produces a .png image of the visualization with the new submission highlighted.

This tool can easily be applied to new sets of submissions as well; it is designed to let a grader cluster their own set of submissions, not just those from CS61A, project 2. While this script requires a significant amount of time due to the pairwise comparison, it is only a preprocessing step, meaning the overhead is a one time cost. Once the huge set of submissions are pairwise compared, perhaps run overnight, the tool can be used to analyze submissions and where they lie among others, as well as take in new submissions for future semesters. As demonstrated in the analysis portion of this paper, the tool can be used to identify common mistakes, different types of solutions or ways to solve the problem, and even help us gain a better understanding of different design decisions.

5.2 Web Application Tool

While visualizing clusters can be informative and useful, we wanted to make this process more usable for providing feedback or grades back to students and instructors. We created a web application that allows graders to grade submissions in a more streamlined fashion. The tool allows for clusters to be tagged with specific feedback that will be sent back to the students. Graders grade one submission at a time, but when they come across a submission in a cluster that they have already seen, those same comments will come up as suggested feedback remarks and grades. In this way, specific feedback that applies to multiple submissions within the cluster could help expedite the code review process. Furthermore, the tool provides submissions within the same cluster in a chronological order to organize and simplify the process for the grader. Now the grader is able to grade similar submissions at one time, focusing on a particular structure of code and really nailing down the feedback for those submissions. This approach is aimed at helping the grader grade and provide feedback, by making the process more consistent and efficient.

5.2.1 Application Design

In the simplest case, a grader would upload the submissions they were attempting to grade to our web application. For each submission, the automated tool would run on the backend to convert the code to JSON, calculate the edit distances between older semesters' submissions and the current submission, and finally retrieve the cluster id that this particular submission belongs to. The grader would then be redirected to the grading page, displaying a file viewer on the right, and the ability to add comments and a score on the left, as shown in Figure 5.2. In order to choose comments from those that have been already used, we offer a simple checkbox list, which the grader can update. Additionally, we hope to add the functionality to let the grader specify the line numbers of where the composition errors occurred. This part would be per individual mistake, but not be saved for priority queue duplication purposes. A screenshot of the live application is shown in

Figure 5.3. Finally, when the grader clicks the submit button, “Send to Student,” the comments as well as the chosen score, are emailed to the student’s login.

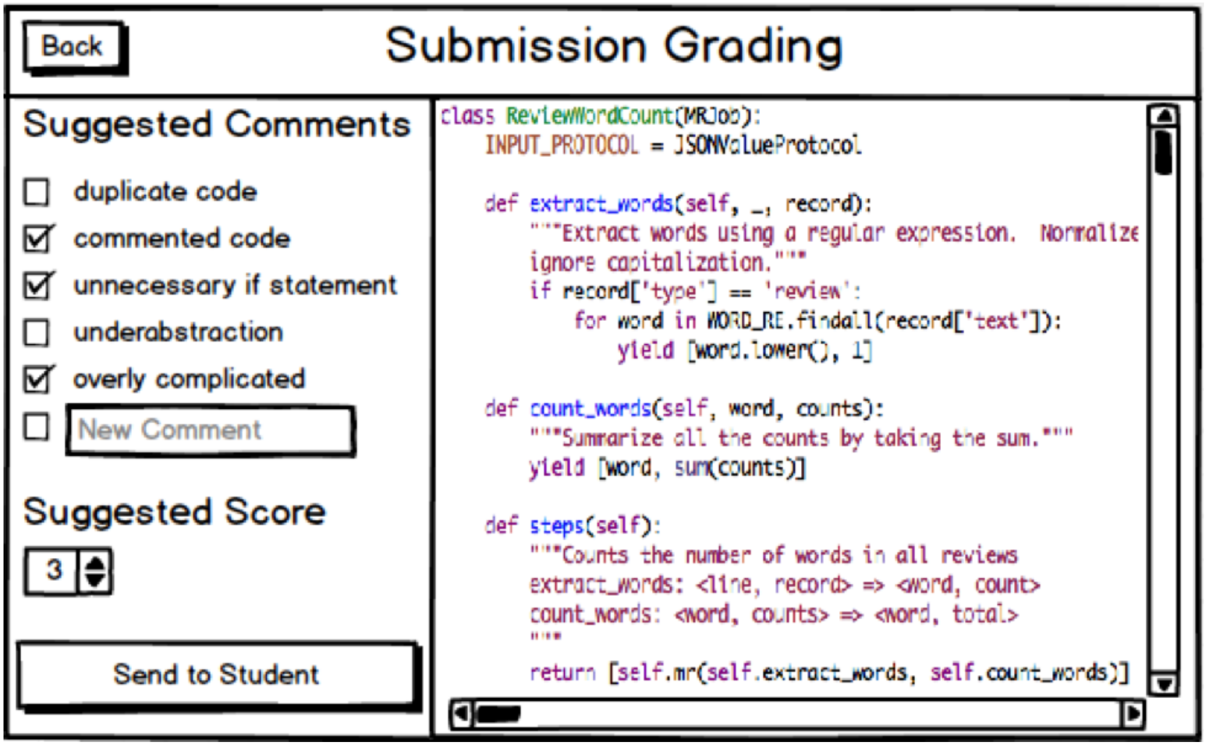

Figure 5.2. Mockup of Web Application

5.2.2 Priority Feedback

As the grader grades submissions from particular clusters, the web application saves the comments and scores that the graders are giving to the student per cluster id. The frequency of the comment per cluster id is kept in a priority queue, which is then accessed to show the top N comments as “suggested comments” when grading a submission from the same cluster. While we do not deal with semantic disparity, this is obviously a place for improvement. The suggested score is also calculated by taking the average of all submission scores in that cluster. While the suggested score may be a fractional quantity, the grader is required to choose from the grading scale 1-3, as determined by the class.

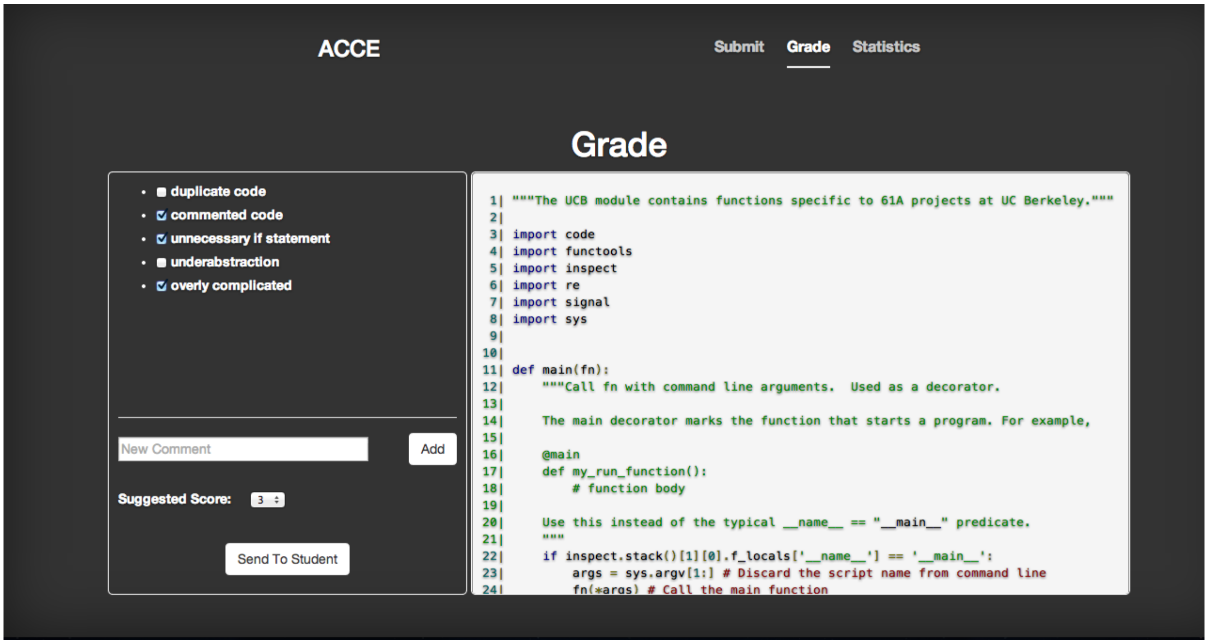

Figure 5.3. Screenshot of Live Application

5.2.3 Usability Testing

Dealing with the same data from CS61A's project 2, we focused on one particular function for our usability tests: `find_centroid`. We were interested in determining how efficient this process was with respect to the currently implemented process that graders currently used to grade code based off of composition. Additionally, we wanted to know how consistent graders were while using our system, as we have already determined there is a lack of consistency with the current method.

5.3 User Study

The final implemented prototype was a web application, built in Django and Python, that allowed a grader to grade code based off of composition through a web interface, used clustering to group similar submissions and provide suggested feedback, and implemented the design as described in detail above. This prototype was built with the help of an undergraduate student, Daniel Kang. An image of the prototype is documented in Figure 5.3.

Although the grading was not entirely automated, the grader was able to grade similar code submissions (based on composition and structure), allowing them to focus on similar mistakes at one time, as well as provide the feedback to similarly-composed code fragments. The user study allowed us to better understand and measure, both qualitatively and quantitatively, how well the prototype helped the grader grade code submissions based off composition. While our main concern is in comparing the efficiency of our system to the currently implemented code review process, we wanted to assess the overall quality of the prototype and assess all aspects of potential benefits. Other than efficiency of grading, the user study was designed to measure accuracy, and consistency of grading.

To test the prototyped web application of the new code composition review system, we ran a user study with 6 current or former graders for CS61A. Graders were recorded while they graded a series of coding submission fragments, and the recorded data, along with the comments, scores, and other user actions were analyzed to attempt to measure quality of the ACES prototype. The user study is described in more detail below.

5.3.1 Study Design

The first step involved setting up the environment. We created 14 code fragments (functions) in Python, with various composition mistakes. We then randomly duplicated one code fragment to make 15 total, which were then pre-loaded into the web application for grading.

The subjects were broken down into two major groups: the control and experimental group. The control group consisted of 3 graders, who were asked to use the current system of manual code inspection to grade these coding fragments based off of composition, while the experimental group used the prototype described above. Each group was then asked to perform the same steps.

After signing the consent form, the grader was presented with each of the 15 code segments. All code segments except for the two duplicates were presented in random order to the control group, and by cluster for the experimental group. The two duplicates were presented at the beginning and end of all submissions for the control group and the beginning and end of the particular cluster group those submissions identified with for the experimental group.

The grader proceeded to grade each code segment as if they were grading a student's project submission. They commented (provided feedback) on any composition errors in the web application itself, and chose a score from 1 to 3 for each submission. We recorded their on-screen activity using Quicktime's screen capture tool. Each group was allowed to switch between submissions as they saw fit, in case they were interested in amending a previous comment/feedback remark or score. All recordings were erased immediately after the analysis was performed.

5.3.2 Post-Study Questionnaire

In addition to the experiment detailed above, we had each subject answer a short post-experiment questionnaire, asking them the following:

1. Rate the grading process on a scale from 1 to 5 (1 terrible, 3 average, 5 fantastic).
2. What problems if any, arose during your grading?
3. How accurate did you think your grading was on a scale from 1 to 5? (1 not at all, 3 average, 5 extremely).

Note: in order to get a more detailed response, we asked the user to rate on a scale that included half points for both question 1 and 3.

5.3.3 Results

From there we analyzed various qualitative and quantitative metrics. While we only had three people per group, (control and experimental), we were still able to see some differences among each metric.

Ease of use

As predicted, we saw many more complaints of "problems" from the prototype than from the control group, as this prototype was not fully developed into an industry-standard application. For the second question we received 5 different issues for the prototype compared with 2 from the normal code review system. Obviously, there are a few explanations. This was for an untested prototype: there were a few UI bugs and usability features we had not fully implemented or tested. We received remarks such as "difficult navigation," "lost part of a comment because of a miss click," and more generally "UI bug." Another reason for the higher number of problems was that subjects in the control group had already used that grading system before, as it is the currently used system. There was no learning curve, as there was for the experimental group.

Finally, we saw that the opinions among the two groups on the overall feelings of the grading process were rather similar. The ratings we received for the grading process for the prototype were an average of 4.334, while those in the control group responded with an average rating of 4, with all scores for the prototype being equal or higher than the control code review system. The small difference may be due to a bias in the added functionality: users in the experimental group noticed that coding submissions were ordered in a particular way (based off of similarity).

Efficiency

We measured the time it took to grade all 15 coding fragments for both groups and how many user actions (clicks, types, etc) they performed to accomplish the grading. Despite the various complaints about our prototype, we saw that there were significantly less user actions (which included scrolls, clicks, character typing, and other movements on the part of the subject), while grading any particular submission while using our prototype. There were an average of 10 fewer user actions for the prototype than the control code review system. The reason it was not a larger disparity came from the fact that a majority of the graders in the experimental group wound up returning to past assignments to regrade then, causing more clicks.

Overall, the prototype system took graders an average of 32 minutes to grade all 15 submissions, while the control code review system took the graders 35 minutes. This is not a huge difference, and since there were not very many subjects in this user study, this is probably not statistically significant. It does, however, give us a glimpse of the possibilities in increased efficiency, as we already know our system requires far fewer user actions to accomplish the grading.

Accuracy

We carefully crafted each of the coding segments with various composition errors. However, there were a few composition errors that were identified during the experiment, which we also included in the subset of mistakes made per problem. Using this set of correct "mistakes graders should identify", we were able to vaguely gauge the accuracy of the grader, in a way different than scoring on a scale from 1 to 3.

Figure 5.4 shows the accuracy of the control and experimental group per coding segment (labeled 1 - 15). Coding segments where the graders identified all mistakes (segments 2, 3, 4, 12) or there were 0 mistakes total (segment 6 - 10) were ignored. That left only those coding segments that had some mistakes missed by the grader or some other discrepancies. (While we will get more into this in the next section, it is important to note that code segments 1 and 15 were in fact the same coding segments).

Figure 5.4 shows that our prototype does in fact increase accuracy. Although in most cases, the graders from both groups missed the same number of mistakes, in a few cases, the graders from the experimental group identify more mistakes than those from the control group. For example, in code segment 13, one grader in the control group completely missed the only mistake in the code. The experimental group was primed for this exact mistake by the time they saw this coding submission, as the other coding submissions they had just graded made the exact same mistake because they were in the same cluster. Similarly, in

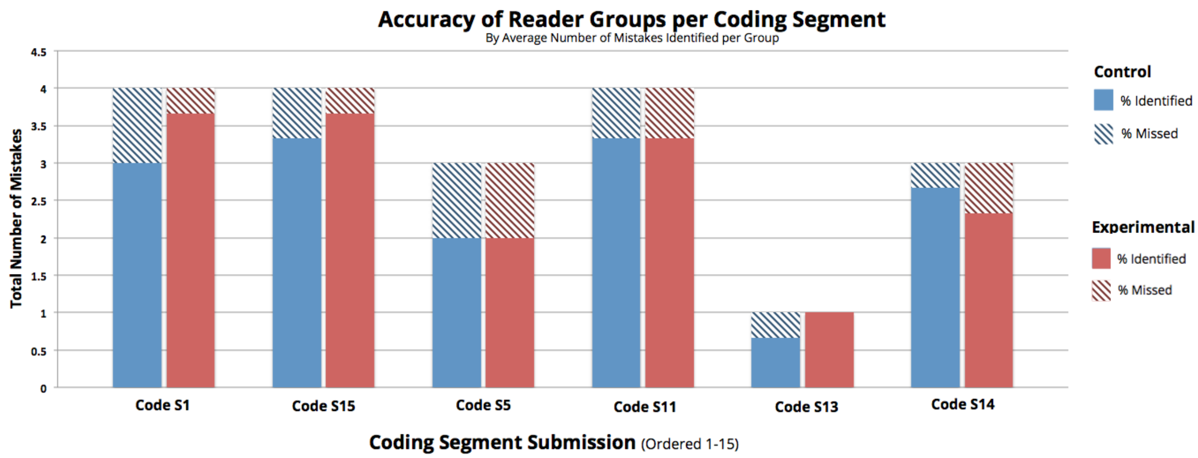

Figure 5.4. Accuracy, average percentage of mistakes identified out of total mistakes, of each group of graders, control and experimental

code segment 1, all graders from the control group missed the mistake of calling a function more than once with the same parameter, or a repeated function call, while only one person from the experimental group did.

While we were hoping to compare how the graders rated themselves on accuracy, post-survey question 3, to the above accuracy measure, it seems that both groups of graders said they graded accurately at a round 4 on the 1 through 5 scale. Thus no interesting insight could be drawn about the self-grade or the effort on the part of the grader. It seems that most of them missed at most one mistake per grading segment anyways, which seems accurate to the 4/5 rating.

Consistency

In order to test how consistent any grader was while grading, we decided to put in a duplicate code submission, in hopes that the grader would identify the duplicate or at the very least grade both in the same way. We made sure to present the duplicate at a fair distance from the original, in order to re-enact similar situations within a normal grading environment (including the prototyped environment). From each group, only one grader missed the duplicate, showing that the prototype did not necessarily help to distinguish duplicate code for the human eye. However, in terms of the accuracy from above, we see that one grader from the control group actually graded these two submissions differently, identifying one additional mistake after viewing the same code again. We propose that because the grader was not focused on a particular error, as the clustering grouping in the experimental group was, the grader missed the mistake completely, but was unable to effectively find or remember the submission that had the same error.

This only measures intra-grader consistency: the consistency of one particular grader over time. However, an additional analysis could be run to show consistency among graders, inter-grader reliability, by taking the score they gave each of the coding segments, similar to what we did in the initial stages of our research. While it would be interesting to look at the effect the prototype has on this particular measure, we were unable to compare scores due to a loss of data.

5.3.4 Discussion

As we were limited to the number of participants we listed in our IRB application and the number of graders willing to take part in the study, we were only able to run the user study with a total of 6 past graders. The user study did not have enough participants to be deemed statistically significant, but can shed some light into the possibilities of such an application. The differences in measurements are quite small, and often didn't lead to the conclusions we had wished to make. From our observations, the application definitely lowers the amount of user clicks and helps with accuracy to some extent, but had no effect on the consistency of grading, and was harder to use. Overall, we believe the idea of priority comments suggested based off of use for particular clusters could go a long way if implemented and tested more rigorously.

Chapter 6

Future Work

The efficiency of both of our tools, in terms of the one-time preprocessing step, is extraordinarily slow and could use improvements. The main bottleneck is in the pairwise comparison that has to be performed to get a similarity metric between all pairs of submissions for clustering. This step could be optimized in several ways, including reducing the number of comparisons. Improving the efficiency directly leads to a more scalable solution, as this automation or streamlined grading is meant for massive courses.

Additionally, the process can be greatly generalized and extended to work with a variety of programming languages and classes. Right now, the visualization tool can be run with any set of Python submissions, but are required to be smaller functions with skeleton code. The process is easily extended to work with larger code segments, but is not currently implemented. Both tools only work with Python code. However, for the visualization tool, one simply needs to convert their coding submissions into AST of the exact format for our tool. This could be automated, by using lower level implementations of languages.

Finally, while our web application code review system was implemented, it was just to act as a prototype, specifically for the user study written in this report. Adding the functionality of our tool to the existing code reviewing system that is used in CS61A would be ideal, as it is one slightly adapted from Google's own code reviewing system.

Chapter 7

Conclusion

Automating the subjective process of grading code based off of composition is extremely difficult. Instead, focusing on ways to help expedite the process was proven to be a much more realistic approach. Clearly, there are common trends among submissions which can be leveraged to make grading a more efficient process, especially when working with a dataset of such size, as MOOC courses often do. Our tool, even now, provides interesting insights to general trends, as well as specific ones that allow readers to stay more consistent when grading. This is especially true if done by function rather than as a whole submission. Applying feedback and scoring to submissions within the same cluster, from our preliminary analysis, seems to be a reasonable way to grade coding submissions.

7.1 Potential Applications

The research done here has a variety of applications. The main purpose is that we can use clustering to streamline or even partially automate the code review process. Grouping together similar coding submissions means readers can focus on one type of solution at a time, and really nail down the common mistakes. Furthermore, they can use similar comments and scores in their feedback. One can even go so far as to claim that choosing representative nodes from each cluster can allow a reader to grade just a few submissions in a cluster and propagate all feedback and the score to all other submissions within the same cluster.

Additionally, we can use this knowledge for peer-pairing: pairing up students with different style habits would allow the propagation of good habits, or pairing up students with similar style might allow them to understand each other's code more quickly. The clustering technique has the unique advantage of being able to show us pairwise similarity between students and their coding style.

One interesting application of the visualization tool would be to show how submissions change over time. Classes that allow multiple submissions of the same assignment could benefit from the automated visualization of coding submissions, by watching how the clusters change over time, hopefully showing improvement.

Finally, clustering student submissions in this way gives the instructor a better idea of the type of solutions for a particular problem. In other words, they can identify common mistakes among their students, and it allows them to more easily pinpoint student confusion. Even for those solutions that don't have any stylistic errors, we might see different ways of solving a problem, which could lead to interesting ways of looking at and studying design choices. In summary, the automated visualization tool can be used to identify common mistakes, different types of solutions or ways to solve the problem, and even help enforce consistency among graders.

Chapter 8

Appendix A

Number of Repeated Function Calls: 3

```
for x in polygons:
  lat += find_center(x)[0]*find_center(x)[2]
  lon += find_center(x)[1]*find_center(x)[2]
  a += find_center(x)[2]
```

Number of Lines of Repeated Code: 3

```
for x in polygons:
  lat += find_center(x)[0]*find_center(x)[2]
for x in polygons:
  lon += find_center(x)[1]*find_center(x)[2]
for x in polygons:
  a += find_center(x)[2]
```

Length of Variable Name: 3

```
lat += find_center(x)[0]*find_center(x)[2]
```

Length of Line: 42

```
lat += find_center(x)[0]*find_center(x)[2]
```

Length of Code (number of lines): 5

```
def find_centroid(polygons):
  for x in polygons:
    lat += find_center(x)[0]*find_center(x)[2]
    lon += find_center(x)[1]*find_center(x)[2]
    a += find_center(x)[2]
```

Meaning of Variable Names (English Word): False, 0

```
lat += find_center(x)[0]*find_center(x)[2]
```

Number of Lines of commented out code: 2

```
# for poly in polygons:
for x in polygons:
  lat += find_center(x)[0]*find_center(x)[2] #*find_center(x)[1]
  lon += find_center(x)[1]*find_center(x)[2]
  a += find_center(x)[2]
```

Code Coverage (# of lines unreachable): 1
Code Coverage (% of code covered): 0.8 (80%)

```
x = 2
if x == 3:
  y = 2
else:
  y = 3
```

Number of Redundant if Statements: 2

```
if x = 2:
  y = 3
if x = 2:
  z = 3
```

References

[1] Aggarwal, V., and Srikant, S. Principles for using machine learning in the assessment of open response items: Programming assessment as a case study, 2013.

[2] Basu, S., Jacobs, C., and Vanderwende, L. Powergrading: a clustering approach to amplify human effort for short answer grading. *TACL* (2013), 391–402.

[3] Bridge, P., and Appleyard, R. A comparison of electronic and paper-based assignment submission and feedback. *British Journal of Educational Technology 39*, 4 (2008), 644–650.

[4] 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.

[5] DeNero, J., and Martinis, S. Teaching composition quality at scale: Human judgment in the age of autograders. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, ACM (New York, NY, USA, 2014), 421–426.

[6] Gephi, http://gephi.org/.

[7] Gitchell, D., and Tran, N. Sim: A utility for detecting similarity in computer programs. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '99, ACM (New York, NY, USA, 1999), 266–270.

[8] Glassman, E. L., Singh, R., and Miller, R. C. Feature engineering for clustering student solutions. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, ACM (New York, NY, USA, 2014), 171–172.

[9] Heaney, D., and Daly, C. Mass production of individual feedback. *SIGCSE Bull. 36*, 3 (June 2004), 117–121.

[10] Huang, J., Piech, C., Nguyen, A., and Guibas, L. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *Proceedings of the 16th Annual Conference on Aritificial Intellgence in Education*, ACM (2013).

[11] Luxton-Reilly, A., Denny, P., Kirk, D., Tempero, E., and Yu, S. On the differences between correct student solutions. In *ITiCSE*, ACM (2013), 177–182.

[12] MacWilliam, T., and Malan, D. J. Streamlining grading toward better feedback. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, ACM (New York, NY, USA, 2013), 147–152.

[13] Mory, E. Feedback research revisted, 2004.

[14] Pep8, http://www.python.org/dev/peps/pep-0008/.

[15] Piech, C., Huang, J., Chen, Z., Do, C. B., Ng, A. Y., and Koller, D. Tuned models of peer assessment in moocs. *CoRR abs/1307.2579* (2013).

[16] Piech, C., Sahami, M., Koller, D., Cooper, S., and Blikstein, P. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, ACM (New York, NY, USA, 2012), 153–160.

[17] Pyflakes, http://www.divmod.org/trac/wiki/divmodpyflakes.

[18] Pylint, http://www.logilab.org/857.

[19] Reily, K., Finnerty, P. L., and Terveen, L. Two peers are better than one: Aggregating peer reviews for computing assignments is surprisingly accurate. In *Proceedings of the ACM 2009 International Conference on Supporting Group Work*, GROUP '09, ACM (New York, NY, USA, 2009), 115–124.

[20] Saikkonen, R., Malmi, L., and Korhonen, A. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ACM (2001), 133–136.

[21] Schleimer, S., Wilkerson, D. S., and Aiken, A. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, ACM (New York, NY, USA, 2003), 76–85.

[22] Singh, R., Gulwani, S., and Solar-Lezama, A. Automated semantic grading of programs. Tech. rep., MIT/Microsoft Research, 2012.

[23] 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 (2011), 83–92.

[24] Thorpe, M. Assessment and third generation distance education. *Distance Education 19*, 2 (1998), 265–286.