Learning Analytics

Austin Gibbons

December 2011

1 Introduction

Computer Science educators are faced with the trade-off between the amount of time available and giving students individual attention. In many classrooms, particularly in the K-12 grades, it is not possible for a single instructor to give as full of feedback to every student in an entire classroom as they would if they were teaching in a one-on-one setting. We hope to develop tools to automatically detect the evolution of student programming methodology over an entire computer science course, and thus offer a way to quickly characterize the progression in programing style of a student that instructors can use to provide feedback, assistance, and swiftly target the specific needs of their class. We are specifically interested in studying novice level programming courses due to the abundance of data, the large and general applicability of the study, and to capture more fundamental "ground truths" of programming education versus the many influencing factors that enter at the higher level computer science courses.

2 Data

This work extends from work by Piech et al [1] which analyzed patterns in single java project. Using the data files from that work, our goal was to do a similar analysis over the progression of the class by analyzing projects the students developed from week-to-week. We choose to analyze CheckerboardKarel.java, Pyramid.java, Breakout.java, Hangman.java, and Yahtzee.java. These are significant implementation files from the first five assignments for cs106a, the introductory programming course at Stanford University. We used a sample of 61 students from the same quarter who submitted all five assignments. When the student develops, every time they compile their code, a copy of the *.java files are sent to an external server. This allows us to observe the development process of a student as they build their code.

In order to model the progression of a student throughout the class, we took the individual features and chose some aggregating functions to represent the student's changes across the program. This required some exploration of different features. We condunceted a preliminary study over commenting features, and we observed that just summing the changes across submissions was insufficient to separate the students, and that generally an individual metric was insufficient to fully capture the patterns we would later observe. We found a good representation of the changes to be the average the feature was changed (average), the largest difference between any two submissions (range) and the total number of times a feature changed between submissions (count). We then created the <average, range, count> tuple for each feature, thus creating a new feature list for each student that includes these aggregations for several consecutive projects the student built.

3 Feature Selection

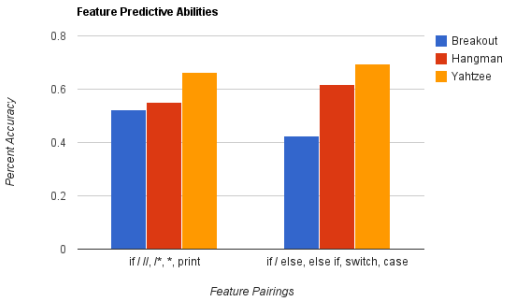
Part of the challenge in performing analysis in student code is ensuring that our ad hoc assumptions do not limit the scope of the investigation. As part of the study, we needed to perform feature selection to make comparisons across specific aspects of students code. For example, our intuition may tell us that

there will be a strong relationship between the number of *if* statements and the number of *else* statements as the latter requires the former, but it may be that for some assignments, many students choose to use *switch* statements instead. To address the question of which features are rightly coupled, we attempt to form classification over a subset of the features and determine if we can correctly classify an external feature. The canonical approach would be to compute the conditional probabilities and find strong correlations, but as a machine learning exercise we took a different approach.

As part of our development, we built a classification system to address the following question: "Given student grades as labels and the features we have collected, can we predict students' grades without ever looking at their code?" Unfortunately, we were not given permission to access student grades in sufficient time to execute this study, as we had to appeal to Stanford's Human Subject Research Institutional Review Board for access to sensitive information. We shelved this project for next quarter, but instead asked "can we predict feature characterization from a subset of the total features?". To this end, we pigeonholed students into being a "frequent" or "infrequent" and awarded labels 1 and 0 respectively. We then conducted experiments over different combinations of features to determine which were good predictors. We tried several classifiers. We found the linear relationship between some features caused non positive definite covariance matrices for quadratic classifiers, so we used a linear classifier to be able to analyze all of the feature subsets in which we were interested.

While the prediction rates are not high enough to make claims of strong predictive ability, we can nevertheless get a sense of good feature selection from the predictive ability of our classifier. We can observe that different assignments produce different predictions, and this allows us to better rationalize about the relationship between individual features (such as the results in section 4.3). When we observe the trends in these sections, we can do so knowing the more global relationship between the features we are comparing and can account for that bias.

As an example, we compare the predictive abilities of *if* statements relative to *print* statements and $//, /*, *$, comments versus prediction on *if* statements, *elseif*, *else*, *switch*, and *case* statements.



We can make several conclusions - first note that I have chosen the assignments from weeks 3, 4, and 5 of Stanford's introductory java course. The assignments from earlier weeks did not have enough usage of these feature to warrant comparison. We can see the general increase in performance of our prediction, and this is derivative of the increased number of lines of code in the assignments as the students struggle with more complex concepts.

We can further deduce that there is a relationship amongst our randomly chosen features, but it is not as pronounced as the relationship amongst features we suspected would be related (the second set of bars pertaining to branching statements). We can observe the introduction of switch case statements in lecture by the jump in predictive ability from the blue bar to the red bar, going from essentially random to meaningfully high accuracy. In section 4.3 we discuss how some students abandoned *switch* statements in favor of using *elseif*, but here we can see our ability to recognize students making with regards to branching statements increases. We can infer that there is a relationship between understanding different branching techniques and choosing which branching statements to use from our ability to predict more accurately in light of students more evenly distributing themselves across the programming methodology.

4 Programming Methodology Analysis

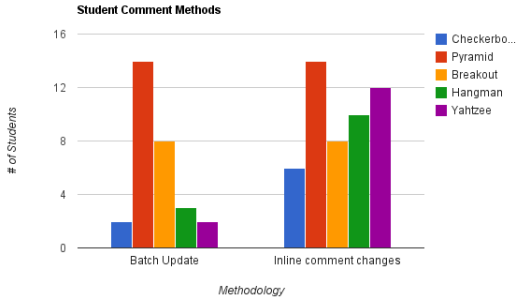
We elect Course Assistants from the class as domain experts to assist in the analysis of the data. Given our list of features, we perform k-means clustering on the students, and have the domain experts perform manual analysis on the cluster centroids. We tried different cut-offs for the k values and discovered as few as four features already created clusters with only one or two students. This is indicative that there were only two or three main cluster bodies, so we conducted analysis at two, three, and four clusters. We made the simplifying assumption that the feature centroid was representative of the members of that cluster. This may not be perfectly true (and is something we are investigating) but an ad hoc analysis of the features saw this to be generally true. By observing the size of the clusters and the feature centroids, our domain experts were easily able to realize the underlying patterns of student programming methodology.

We performed this analysis on the comments the students added to get an estimate for the documentation practices, and the java code they wrote to get an estimator for the amount of code written. We took our clusters and found the centroid, and stated that this was the pattern that each student who fell into this cluster used in their program. We then used the relative sizes of the clusters as an indicator for the number of students who programmed this way. We performed analysis of the centroids without revealing the number of students in each cluster in order to avoid pushing the results in a desired direction.

4.1 Comments

We count the number of different comment lines (//, /*, *, */) and use this as a feature set. These features give us an indication of the preference for block comments, line comments, and commenting frequency. Some interesting patterns that arise in the are

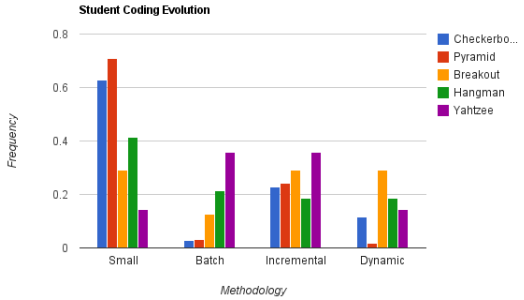
1. How the frequency of students who comment all at once (Batch update) changes
2. Students who use line comments for bookkeeping



We can observe a natural progression in the maturity of the commenting style as more students abandon the idea of writing all their code and entering all the comments at the end. CheckerboardKarel being the exception as the students had not yet had any feedback about commenting procedures, so many did not add to the comments (some were written into the assignment by the class staff). We can additionally see an increasing preference for using inline comments. These are used especially for bookkeeping data, and get changed often throughout the program with few of them making it into the final write-up, and many only persisting as the programmer develops. Empirically, we can see that students tend to morph some of their inline comments into block comments, usually as method headers.

4.2 Lines of Code

Mirroring our comment features, we also cluster by using the <average, range, count> of lines which are not comments, namely lines which are java code. We perform a comparison aimed at separating "Tinkerers" from "Planners", the former preferring Trial and Error, and the latter having a preference to have a more detailed plan before they begin coding. We classify the students as performing large "batch" updates, small updates, incremental updates, and updating dynamically. A student can fit into multiple categories, for example updating in small increments versus batch increments. We observe several interesting trends in this model:



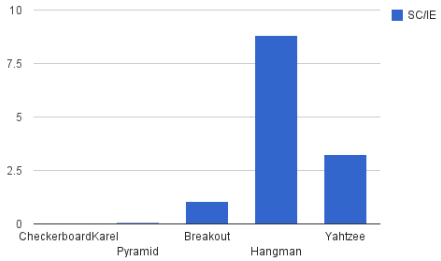
It is important to note that CheckerboardKarel and Pyramid both have generally much smaller submissions, as they are much shorter assignments.

The increase in total line count difference ("Batch") noticeably increases with each submission. The programs themselves are not increasingly large projects as to require that much more code, but rather this is indicative of the conceptual challenge the students face - as young programmers they often require unnecessary verbosity relative to what an experienced programmer requires to solve the same problem. They are required to handle increasingly complex data structures and increasingly complex algorithms and this is represented by their increase in verbosity, as many students chose naive and lengthy solutions over concise solutions. For example, many students in creating Yahtzee wrote long and inefficient methods to perform finding three of a kind, four of a kind, full house, small straight and long straight, failing to recognize the simpler solutions for finding $matchNofaKind()$ and $matchStraight()$.

Additionally, observe that students who code in a dynamic way (many more commits relative to the size of the commit) initially jumps at the first large project Breakout, and then decreases as the quarter progresses. This may be representative of an increase in programming skills, as the students need to use less trial and error. Additionally, we observed an increasing trend in the frequency and size of code deletion, which is indicative of students recognizing good decomposition and code reuse when they re factored their code as part of their "style" grade.

4.3 If-Else versus Switch-Case

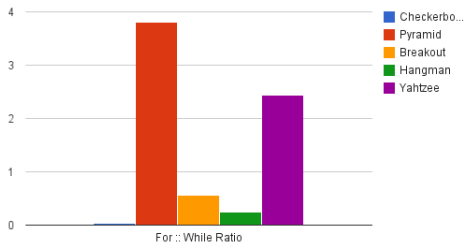
We can also observe how students respond to new topics. For example, they were exposed to the idea of a switch statement in lecture before the Breakout assignment, and after getting feedback on its use from the graders, many more adopted it in Hangman. We can see that its use dropped off in Yahtzee, showing that while many students did prefer it, some did reject it after their experimentation. By analyzing the specific students in each cluster, we can survey the students asking why they abandoned using switch statements in favor of the usually messier if else chaining. This provides educators with more specific and targeted feedback, and will let them adjust their teaching style to accommodate the needs of specific individuals. We can observe this trend by looking at the ration of students who use $if/else :: switch/case$ easily:



4.4 While loops versus For loops

As another example we can see the introduction of for loops in the second assignment, and the quick stabilization as students realize that both *while* loops and *for* loops are valuable. What this graph additionally shows is how to infer information not just about students, but also about your assignments. When students demonstrate a mastery over simpler topics, it is a larger statement on the problem set than the students that the fifth assignment involved many more students using a large amount of *for* loops, itearting over the five dice and the thirteen scoring categories. Performing this analysis over the full set of features allows an instructor to check his expectations of the

programming assignment in addition to what the students are learning. Note that this graph counts students who use a large number of *for* loops and *while* loops as having chosen to use both.



5 Conclusions & Inference

The most important thing to infer from this study is the simple ability to automate qualitative analysis. We hope that other researchers may find success in running the same methodology over their own students code, even if they observe different trends.

We hope that we can accurately characterize the relationship between features and offer insight into the programming paradigm shift the student body takes as they learn new programming tools.

We believe these education analysis tools will allow educators to target the individual needs of their students and offer the ability to assess which students are retaining the material. We can provide educators with specific students who chose different programming methods, and this enables the instructor to get specific, targeted feedback. We additionally hope that the results offer educators of introductory programming classes a solid model of the typical progression a student body takes, so that they have something other than an ad hoc evaluation of their own teaching methods.

6 Acknowledgments

This work was done in conjunction with the Learning Analytics group in the Transformative Learning Technologies Lab. The author conducted his experiments independently. He would like to thank Professor Paulo Blikstein, Marcelo Worsley, Mustafa Safdari, and Tarun Vir Singh for their feedback on his ideas and progress.

References

[1] Chris Piech, Mehran Sahami, Daphne Koller, Stephen Cooper, and Paulo Blikstein. Modeling how students learn to program. 2010.