Inferring Use Cases from Unit Testing

Jaime Spacco
University of Maryland
College Park, Maryland 20742
jspacco@cs.umd.edu

Titus Winters and **Tom Payne**
University of California, Riverside
Riverside, California 925216
{titus,thp}@cs.ucr.edu

Abstract

We present techniques for analyzing score matrices of unit tests outcomes from snapshots of CS2 student code throughout the development cycle. This analysis includes a technique for estimating the number of fundamentally different features in the unit tests, as well as a survey of which algorithms can best match human intuition when grouping tests into related clusters. Unlike previous investigations into topic clustering of score matrices, we successfully identify algorithms that perform with good accuracy on this task. We also discuss the data gathered by the Marmoset system, which has been used to collect over 100,000 snapshots of student programs and associated test results.

1. Introduction

A common data type for educational data mining (EDM) researchers to work with is the *score matrix*, a collection of student scores on a set of questions. In last year's AAAI EDM workshop, two papers focused exclusively on this type of data (Barnes 2005; Winters *et al.* 2005). In the first of these the input matrix came from student scores on online quizzes, in the second the scores came from in-class tests and quizzes. Both looked to provide an unsupervised learning method for separating questions into related groups based on the data in the score matrix. As shown by Winters and Payne, this can be a surprisingly challenging problem, depending on the source of the data. Determining what types of score matrices can be meaningfully clustered by topic is an interesting and useful line of investigation for the EDM community.

The score matrices studied thus far by the EDM community have consisted solely of quiz or exam data. However, the computer science major also has a substantial programming component that must be evaluated outside of a quiz setting. Programming assignments are typically evaluated using a suite of test cases or test data, often written in the form of *unit tests* using a unit test framework such as JUnit (Patterson, l Kölling, & Rosenberg 2003).

As with decisions about quiz and test questions, decisions regarding how to test programming assignments are routinely made with little or no quantitative feedback, even for projects that are re-used from previous semesters. Ideally, we'd like to collect and analyze unit test outcomes for programming assignments, in order to discover "use cases" (related unit tests). Identifying use cases can have many benefits to the instructor. For example, the total number of use cases may give a measure of the complexity of an assignment (since each use case should represent a different underlying concept to be tested), while identifying closely related test cases can reveal redundancies in the test suite that can be trimmed in future semesters or scaled down in terms of the relative weight toward the final grade on an assignment.

In this paper, we apply EDM algorithms similar to those previously described (Winters & Payne 2005) to extract the underlying features of a programming assignment. The dataset used for this study was collected using the Marmoset system (Spacco *et al.* 2006), an automated submission and testing framework developed at the University of Maryland. Marmoset stores all submissions—not just the final submission—for all students, as well as "intermediate snapshots" captured at the granularity of every file saved. The snapshot and submissions histories harvested by Marmoset for each student can be transformed into an $m \times n$ *score matrix* amenable to EDM algorithms. In this paper, we apply a series of data mining and machine learning algorithms to ten score matrices extracted from two semesters of data from the Marmoset database. We then discover the number of use cases and compare the EDM algorithms with human intuition about which unit tests are related.

2. Data Collection

Marmoset (Spacco *et al.* 2006) is an automated submission and testing system developed at the University of Maryland for testing and providing feedback about programming assignments. Marmoset supports multiple languages, including Java, C, Ruby, and Objective Caml, and in the Spring 2006 semester is being used by six programming courses and over 500 students at Maryland. It is currently being evaluated for future deployment by two other institutions.

Students submit their code to Marmoset's centralized *SubmitServer*, which compiles each submission, runs it against a suite of unit tests, and stores the submission and unit test outcomes in a database. After uploading a submission to the SubmitServer, students can then log into the server and view feedback about the results of running their submission

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

against the unit tests for the assignment. Marmoset also provides a number of pedagogical innovations to control the amount of feedback available to students; however, these features are outside of the scope of the study in this paper. Interested readers may refer to (Spacco *et al.* 2006) for more details.

The unit test suites used by Marmoset are JUnit (Patterson, l Kölling, & Rosenberg 2003) tests designed by the instructor of the course to test the functional correctness of the programming assignment. Note that Marmoset is *only* concerned with functional correctness, i.e. if the specification says that the program should do X, does the program do X or not? Other artifacts of the code that may contribute to the final score for the assignment, such as coding style, comments, and documentation, are orthogonal concerns that are neither evaluated nor recorded by Marmoset and do not figure into any of the datasets used in this paper.

In addition to storing and testing submissions, Java projects can be configured to archive regular snapshots of student work whenever the student saves their files. This is done by a small plugin for the popular Eclipse integrated development environment (Eclipse 2004). The plugin, called the Course Project Manager (Spacco, Hovemeyer, & Pugh 2004), captures a snapshot of a student's files to a central repository every time the student performs a save operation. Because Eclipse constantly performs compilation in the background and underlines syntax errors (similar to the ambient spell-checking features available to most word processors), students do not need to save in order to fix their syntax errors. The practical benefit of this is that around 80% of the harvested snapshots can be compiled and run against the suite of unit tests, yielding a unique and detailed perspective of a student's progress on each programming assignment as well as a large dataset of unit test results.

Marmoset has been in use at the University of Maryland since the Fall 2004 semester. For this study, we have extracted and compiled all the snapshots for five projects in each of two semesters (Fall 2004 and Fall 2005) for a CS-2 course taught in Java by a different instructor each semester. Table 1 details the number of unit tests per project, and the number of snapshots captured by the Course Project Manager plugin. The unit test outcomes are represented as a series of $m \times n$ score matrices for each programming assignment, where m is the number of snapshots and n is the number of unit tests for that programming assignment.

To provide a baseline for comparison of the algorithms we are surveying and human intuition, a human expert (a graduate student experienced with Java programming and unit testing with JUnit) has examined the unit tests for the ten programming assignments used in this study and tried to identify unit tests that can be grouped together because they either exercise approximately the same functionality (and should either both pass or both fail), or exercise incremental components of some larger piece of functionality (for example, such that test case #5 will not pass unless test case #2 passes, but not vice versa).

According to our human evaluator, before any algorithmic groupings were attempted, the F05 datasets broke down into groupings quite easily, while the F04 datasets did not. This may reflect the differing approaches to unit testing of the two different instructors. In addition, our human evaluator may have over-reacted to the lack of natural groupings in the F04 dataset by trying to force unit tests into use-case groupings. Thus, going in to this study, it was known that the unit tests in the F04 datasets appeared to be more independent than the unit tests in the F05 datasets.

The data used in this study represents only a portion of the total data now available. At the conclusion of the Fall 2005 semester, the Marmoset dataset contained over 100,000 compilable snapshots over 3 semesters of CS-2 taught by three different instructors. These snapshots produced produced over 1.2 million unit test outcomes and represented the work of over 250 students. This dataset undoubtedly contains a wealth of untapped information about how novice programmers develop software. The challenge thus far has been finding ways to extract a coherent story from so much rich data. While the dataset has been used for software engineering studies (Spacco *et al.* 2005; Hovemeyer, Spacco, & Pugh 2005), this work marks the first use of Marmoset in EDM studies.

3. Algorithms Surveyed

In this investigation we are looking at a number of unsupervised learning algorithms for reducing our $m \times n$ binary input matrix S into a smaller ($t \times n$) binary matrix. These algorithms come from several different families, most or all of which should be familiar to much of the educational data mining audience. A brief description of the algorithms and how they are utilized in this study is provided below.

3.1 Clustering Algorithms

Clustering algorithms directly solve the problem of identifying which data points belong to the same group. In this investigation we will be using a number of clustering techniques, described below.

k-Means

One of the simplest clustering algorithms is the k-means algorithm (MacQueen 1967). Given the row vectors of S and k, the number of clusters to identify, k-means identifies k *cluster centers*. Each point in the input data is assigned to the cluster center it is closest to, under some distance metric. In this paper we will be using the standard L2 norm.

To initialize the algorithm, the cluster centers are initialized randomly. In each round, the points are assigned to the nearest cluster center, and then the centers are recalculated by finding the point that minimizes the error for each cluster under the given distance metric. This iterative process continues until the points converge to some minima. The algorithm is not guaranteed to converge to a globally optimal set of cluster centers. As a result, random restarts of the algorithm are often used to find better clusters.

Agglomerative Clustering

Agglomerative or hierarchical clustering algorithms initially assign all points to singleton clusters. At each round, the two clusters closest together are linked. Different agglomerative clustering algorithms perform this step in different ways.

F04 (55 students)			F05 (57 students)		
project	# unit tests	# snapshots	project	# unit tests	# snapshots
p14	14	3977	p1	12	287
p16	15	3852	p2	7	1352
p17	11	4265	p3	9	2168
p18	7	1791	p4	12	2379
p20	15	5677	p5	14	2741
Total	62	19562		54	8927

Table 1: CS2 datasets for Fall 2004 (5 projects) and Fall 2005 (5 projects)

Single linkage clustering merges the two clusters with the minimum distance between one point in each cluster (Sibson 1973). *Complete linkage* clustering merges the two clusters with the minimum distance between the farthest points (Defays 1977). *Average linkage* clustering merges the two clusters whose centers are closest together (Sokal & Michener 1958).

Since the desired number of clusters t is already known, the clustering stops when the points have been agglomerated into t clusters.

Spectral Clustering

Spectral clustering (Fischer & Poland 2005) is often implemented using some non-trivial linear algebra, but can be explained as a relatively simple graph-partitioning problem. Given input data composed of n points, form the complete, weighted, and fully-connected graph where the edge between points i and j is given weight equal to the *similarity* of those points under some metric. A spectral clustering into k clusters is done by finding the minimum-weight cuts in the graph that disconnect it into k cliques.

3.2 Dimensionality Reduction

Dimensionality reduction algorithms make up a broad class of unsupervised learning techniques. Although there are other algorithms that could be considered to be members of this class, here we will focus on algorithms that assume a linear model of t factors giving rise to the data in S. That is, algorithms in this family assume the model $UH \approx S$, where U is $m \times t$ and H is $t \times n$. The difference in these algorithms comes from the assumptions that are made about the structure of the factored matrices.

Singular Value Decomposition

Under standard assumptions of least-squared error, Singular Value Decomposition (SVD) is the algorithm that best approximates $S \approx UH$ (Nash 1990). Entries in H (the matrix that maps unit tests to use cases) are theoretically unbounded, ranging $-\infty..\infty$. Geometrically, columns in H identify the t highest-variance orthogonal vectors in the data matrix S.

Since we are primarily concerned with determining which use case each test applies to, there are several ways of interpreting these entries. Use case membership can be assigned based on the maximal magnitude entry in the row vectors of H. This may lead to non-intuitive results: two unit tests v_1 and v_2 can be assigned to unit test n even if $v_{1,n}$ is highly negative and $v_{2,n}$ is highly positive, since only the magnitude of the entries affects group membership.

A method for avoiding this difficulty in interpreting the use cases implied by H is to run k-means clustering on the vectors of H, setting k to t. In doing this we have literally used SVD as a dimensionality reduction and rely instead on k-means to provide the clustering. This provides more satisfactory interpretation, and has shown good results in other domains.

Independent Component Analysis

Independent Component Analysis (ICA) produces U and H similar to SVD, differing primarily in that it may produce vectors that are not perpendicular (Hyvärinen 1999). ICA is more suitable than SVD in situations where the underlying factors are not orthogonal. The same interpretation concerns apply to ICA. We will investigate both the maximal-entry interpretation and the k-clustered interpretation of ICA results.

Non-Negative Matrix Factorization

Non-Negative Matrix Factorization (NNMF) provides a more intuitive model for interpreting the results of the matrix approximation (Lee & Seung 2001). Rather than focusing on vector or geometric methods for approximating S, NNMF focuses on individual entries in U and H, restricting every entry to be non-negative. By further restricting the rows of H to sum to 1, the maximal entries of H provide an intuitive interpretation.

4. Results

Formally, the questions that we are investigating in this experiment are as follows:

- For the algorithms investigated, how does the error in the model depend on t, the reduced dimensionality? Is there an algorithmic method that can be used to determine the proper reduction, or must we rely on heuristic methods like Cattell's scree test (Cattell 1966)?
- When the algorithms investigated have grouped the unit tests, do those groups correspond with human insight on this task? Do the algorithms agree with each other?
- How does the ability of the algorithms to extract meaningful use-case groups depend on the size of the input matrix? Specifically, how many runs of the testing suite are necessary to get good performance?

F04 Data		F05 Data	
Dataset	t	Dataset	t
p14	6	p1	7
p16	6	p2	7
p17	7	p3	4
p18	7	p4	5
p20	5	p5	4

Table 2: Correct t values (number of groupings) extracted for each algorithm

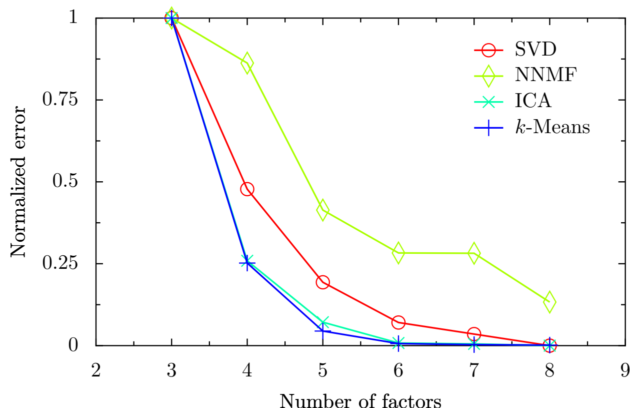

Figure 1: Number of factors vs. error, p3 dataset

Reconstruction Error

If we examine the accuracy of the extracted models as a function of t, we can hope to find some consensus among the algorithms as to the "correct" value for t on each dataset. Although increasing t should always decrease the overall error, for many datasets there is a point of diminishing returns beyond which extracting an additional factor provides little decrease in error. Algorithmically, we can extract the correct t from a plot of t vs. error by looking for (in order of importance):

- "Elbows" on all plots: some value of t for which all plots transition from more-vertical to more-horizontal.
- Low error: if multiple plots have 0 error at the same value of t.
- "Elbows" on most plots: if the majority of plots have an elbow at some t.

To differentiate among ties, stronger elbows, or lower values of t for zero error are given preference. Applying this technique to plots of t vs. error for our ten datasets using k-means, SVD, ICA, and NNMF yields the mapping of datasets to t values shown in Table 2. Sample plots are shown in Figures 1 and 2. Since our SVD implementation is reporting variance rather than reconstruction error, the error is scaled such that the maximum error for each algorithm is 1, allowing for easier graphical comparison.

It is worth noting that this method for discovering the correct value of t differs from most factor discovery techniques in that it is examining the output from numerous algorithms.

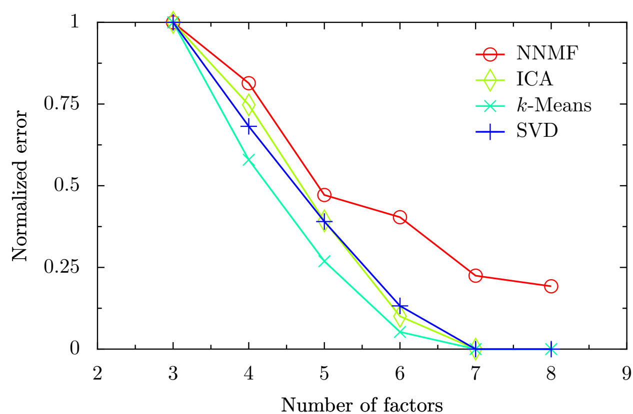

Figure 2: Number of factors vs. error, p18 dataset

Although the techniques used are generally the same as Cattell's scree test (Cattell 1966), the human-judgment behavior is eliminated by taking into account the error rates from multiple algorithms.

Agreement with Human Intuition

Given the values of t listed in Table 2, we can evaluate each algorithm to determine its agreement with the human-generated groups discussed in Section 2. In order to evaluate this agreement, we are focusing on the *precision* and, to a lesser extent, the *recall* of the algorithms with respect to the human generated groups. In order to evaluate these in the context of clustering, we utilize the technique discussed in (Winters *et al.* 2005), where pairs of tests placed in the same group form the basis of the evaluation.

In detail, our evaluation works as follows: each algorithm produces a set of clusters. The output of each algorithm is taken to be the set of all pairs of unit tests that were grouped together[1]. The "correct" answer is taken as the set of all pairs of unit tests that were grouped together by the human evaluator. Precision is the percentage of the algorithm's pairs that are found in the correct answer. Recall is the opposite, the percentage of pairs in the correct answer also found in the algorithm's answer. That is, precision measures how accurate an answer is, while recall measures how complete it is. Given that the evaluator may have grouped a test case into more than one group, perfect recall may not be possible to achieve. In this case precision is the more important of the two statistics, or alternatively we prefer to have false negatives over false positives when grouping test cases.

Precision and recall statistics for all datasets are presented in Table 3. The best algorithm is determined by taking the highest precision, using recall to break ties. If more than one algorithm produces the same precision and recall, all are listed. We are also presenting the probability that two tests chosen at random will be placed together correctly; this is equivalent to placing all tests into the same group.

[1] That is, if one group contains q1, q2 and q3, that group will contribute the pairs q1q2, q1q3, and q2q3 to the final set

Dataset	Best alg(s)	Precision (%)	Recall (%)
p1	avgLink, nnmf	60.00	42.86
p2	nnmf	100.00	10.00
p3	icacluster, kmeans, singleLink, completeLink, nnmf, svdcluster, svd	100.00	87.50
p4	nnmf	46.15	100.00
p5	icacluster, kmeans, singleLink, completeLink, nnmf	66.67	66.67
p14	rand	23.81	
p16	rand	26.67	
p17	ica	42.86	37.50
p18	svd	100.00	28.57
p20	singleLink	23.08	85.71

Table 3: Highest performing algorithms on each dataset

From this table we can see that the algorithm most likely to perform best on this problem is the Non-Negative Matrix Factorization, which is given as the best algorithm in 50% of the datasets. In 30% of the datasets, the single linkage algorithm performs best, which accounts for nearly half of the cases where NNMF is not best. Random, SVD, clustered ICA, k-means, and complete linkage clustering all perform best on two of the ten datasets.

Another method of viewing these results is to average the precision across all the datasets for each algorithm. This result is presented in Table 4.

In this method of evaluation, we again see that the best algorithm on this problem is NNMF. Although NNMF has an average precision of only about 50%, this is significantly higher than average, and a 25% improvement over the next-highest algorithm.

One interesting result is that the average precision for the Fall 2004 datasets is only 35.8% of the average precision for the Fall 2005 datasets. This discrepancy is probably a result of the differences between the instructors who taught the courses. As mentioned in Section 2, the human expert who produced the groupings of the unit tests noted that the Fall 2005 unit test suites contained fairly obvious groupings of unit tests that were nearly identical, while the Fall 2004 unit tests were much more independent. This significant difference in performance is a very positive result: the groupings that are obvious to a human are obvious to the algorithms as well. NNMF does a particularly good job at matching human intuition when there are obvious groupings to be made.

Dependence on Input Size

Another important question is how much the ability of our algorithms depends on the size of the input. That is, can use cases be extracted from test case results with the same precision when the number of test runs (m) is small? Figures 3 and 4 show average precisions for each algorithm as in Table 4 as a function of the number of rows (runs of the test suite) in the input matrix. Each point is the average of 10 randomly-selected subsets of the given size[2] averaged across all 10 datasets.

[2]It is worth noting that the p1 dataset has only 287 entries in it, and thus only affects these plots for small input sizes. All other datasets have over 1000 entries.

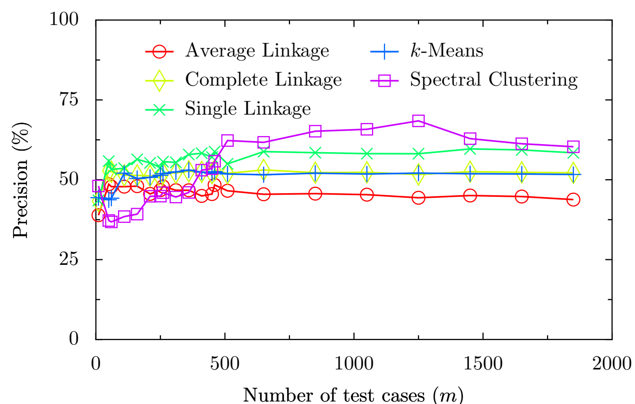

Figure 3: Precision as a function of input size, clustering algorithms

These plots indicate that most algorithms have relatively little dependence on input size. These plots also show that the clustered ICA and clustered SVD algorithms are a good choice for certain types of input, better than would be implied by Table 4. It is somewhat more informative to examine similar plots generated for each individual dataset rather than averaged across the whole collection, to avoid confounding factors like the small size of the p1 dataset. Due to space limitations we are unable to present those plots here. However, we do expect to further investigate some of the unexplained behaviors in these plots in follow-on work.

5. Future Work

There are some important avenues for follow-up and expansion of this study. One of the unaddressed problems in this study is to characterize exactly the dependence on input size. It appears that clustered ICA and clustered SVD may be better performers on smaller datasets, and it may be that NNMF requires 1000 to 2000 rows in S to perform as well as our final results indicate. We have not yet fully explored this matter in this initial investigation. In terms of expansion, there are several other algorithms that should be investigated on these datasets, most notably the Apriori (Agrawal, Imielinkski, & Swami 1993) algorithm for discovery of as-

Algorithm	Avg. Precision	Avg. Precision F04	Avg. Precision F05
nnmf	49.84	18.94	74.56
kmeans	39.96	17.95	61.98
icacluster	39.72	17.45	61.98
completeLink	39.34	16.70	61.98
singleLink	38.95	19.05	58.85
avgLink	36.79	20.08	53.50
svd	35.78	27.17	44.38
svdcluster	35.55	17.45	53.65
rand	31.91	33.10	30.73
ica	31.25	19.74	42.76
speccluster	21.61	17.11	26.11

Table 4: Average precision across all 10 datasets for each algorithm

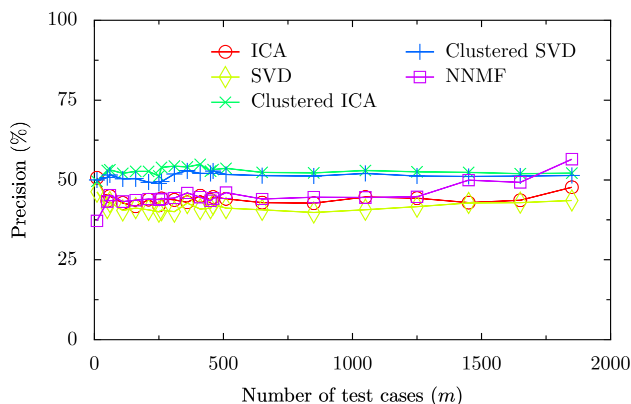

Figure 4: Precision as a function of input size, dimensionality reduction algorithms

sociation rules, and the Q-Matrix algorithm (Barnes 2005). Although an implementation of Q-Matrix was available for this experiment, it was left out of this discussion because of the computational difficulty of performing a discrete optimization in a space with size on the order of 10^8500 for the p20 dataset.

Another important future avenue of research opened up when the data from Spring 2005 was imported into the database. Our work indicates that the instructors from F04 and F05 have written substantially different types of test suites; it will be interesting to compare these test suites with those written by a third instructor.

Thus far, we have measured the EDM algorithms against human intuition assuming that human intuition is correct. We plan to examine the use cases extracted by the EDM algorithms—especially use cases produced by multiple algorithms—in an effort to identify unexpected use cases that our human expert may have missed.

Per-question scores for tests are now being recorded by some courses utilizing Marmoset. Given that the unit test data used in this study seems to partition fairly well, and many of the groups identified by the evaluator are related by an underlying data structure, could the test questions covering those data structures cluster with the unit test data?

5. Conclusions

In this paper we have investigated the ability of common data reduction and clustering algorithms to extract the underlying relationships from a binary score matrix consisting of unit test outcomes of CS2 student code snapshots. This underlying structure allows us to take a collection of unit tests (the columns of S) and determine which of those are testing the same feature of the code. Given such a system, instructors can generate large unit test suites without regard for equal or independent testing, and allow the methods presented here to reduce that suite of unit tests into a smaller collection of use cases and assign grades based on the use cases instead.

What we have seen in this investigation shows that this is a feasible system. It is possible to use a collection of algorithms to determine the intrinsic dimensionality of the data. Given that dimensionality, the best performing algorithm we surveyed is the Non-Negative Matrix Factorization of Lee and Seung, outperforming other algorithms by 25% or more averaged over the full suite of datasets.

An unexpected discovery in this investigation is the difference in accuracy based on which course offering the data was drawn from. All ten datasets were taken from CS2 offerings in the Fall semester, but five were taken from Fall 2004 and five from Fall 2005. Averaged across all of the algorithms surveyed, the F04 datasets only achieve 35% of the accuracy of the F05 datasets. This is possibly due to the instructors differing approaches. Another possible explanation is that the human evaluator who grouped the unit tests noted that the Fall 2004 dataset did not appear to contain as many groupings as the Fall 2005 dataset, and may have tried to force unit tests into use-case groupings.

This work is closely related to the study presented in (Winters *et al.* 2005), but demonstrates conclusively the difference in generative models, and the differing ability to cluster with data coming from those models. In this paper we found a clear ability to do this with relatively high precision on binary data coming out of a unit testing framework. In the previous paper, no methods were conclusively discovered for extracting related questions from matrices of student scores on in-class test questions. Further analysis and characterization of the precise differences between these

two data sources is required. One of the important questions for EDM to answer is: What are the characteristics of data sources (tests, quizzes, unit testing suites, etc.) that generate score matrices that *can* be clustered by topic?

Another feature of this paper is the use of the Marmoset data for research in computer science education and EDM. This data is a unique source of information on the programming practices and behaviors of novice programmers. The dataset is also very large; for example, by Summer 2006 we will have over 200,000 snapshots from four consecutive semesters of CS2, as well as tens or hundreds of thousands of snapshots of CS1 and upper division courses that we have not yet analyzed. Further, since the Marmoset data includes code snapshots throughout the development process for dozens of students, it is possible to begin analyzing that development in ways not previously possible in an attempt to answer questions about how novice programmers program, a hot topic in the CS Ed community for decades (Gruener & Graziano 1978; McCracken *et al.* 2001). While the Marmoset data currently represents only students at UMD, the expansion and adoption of Marmoset at other institutions will shed light on behavior of novice programmers, regardless of institution. The challenge is to develop methods for categorizing and discovering patterns in the code snapshots themselves.

In conclusion, we have presented a fully automatable method for clustering unit test data into groups of related tests. This method raises questions about the underlying models that can be inferred from score matrices and the types of score data can be meaningfully clustered. We hope to see further EDM work on these questions, which have obvious implications for education, assessment, and cognition, in addition to being a useful EDM tool.

References

Agrawal, R.; Imielinkski, T.; and Swami, A. 1993. Mining association rules between sets of items in large databases. *Proceedings of the 20th VLDB Conference* 207–216.

Barnes, T. 2005. The q-matrix method: Mining student response data for knowledge. In Beck, J., ed., *American Association for Artificial Intelligence 2005 Workshop on Educational Datamining*.

Cattell, R. B. 1966. The scree test for the number of factors. *Multivariate Behavior Research* 1:245–276.

Defays, D. 1977. An efficient algorithm for a complete link method. *The Computer Journal* 20(4):364–366.

2004. Eclipse.org main page. http://www.eclipse.org.

Fischer, I., and Poland, J. 2005. Amplifying the block matrix structure for spectral clustering. In van Otterlo, M.; Poel, M.; and Nijholt, A., eds., *Proceedings of the 14th Annual Machine Conference of Belgium and the Netherlands*, 21–28.

Gruener, W. B., and Graziano, S. M. 1978. A study of the first course in computers. In *SIGCSE '78: Proceedings of the ninth SIGCSE technical symposium on Computer science education*, 100–107. New York, NY, USA: ACM Press.

Hovemeyer, D.; Spacco, J.; and Pugh, B. 2005. Evaluating and tuning a static analysis to find null pointer bugs. Lisbon, Portugal: ACM.

Hyvärinen, A. 1999. Fast and robust fixed-point algorithms for independent component analysis. *IEEE Transactions on Neural Networks* 10(3):626–634.

Lee, D. D., and Seung, H. S. 2001. Algorithms for non-negative matrix factorization. In *NIPS*, 556–562.

MacQueen, J. B. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkely Symposium on Mathematical Statistics and Probability*, 281–297. University of California Press.

McCracken, M.; Almstrum, V.; Diaz, D.; Guzdial, M.; Hagan, D.; Kolikant, Y. B.-D.; Laxer, C.; Thomas, L.; Utting, I.; and Wilusz, T. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *ACM SIGCSE Bulletin*, volume 33, 125–140.

Nash, J. C. 1990. The singular-value decomposition and its use to solve least-squares problems. In Hilger, A., ed., *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation, 2nd ed*, 30–48.

Patterson, A.; l Kölling, M.; and Rosenberg, J. 2003. Introducing unit testing with bluej. In *Proceedings of the Information Technology in Computer Science Education Conference*.

Sibson, R. 1973. Slink: An optimally efficient algorithm for the single link cluster methods. *The Computer Journal* 16(1):30–34.

Sokal, R. R., and Michener, C. D. 1958. Statistical method for evaluating systematic relationships. *University of Kansas science bulletin* 38:1409–1438.

Spacco, J.; Strecker, J.; Hovemeyer, D.; and Pugh, W. 2005. Software repository mining with Marmoset: An automated programming project snapshot and testing system. In *Proceedings of the Mining Software Repositories Workshop (MSR 2005)*.

Spacco, J.; Hovemeyer, D.; Pugh, W.; Hollingsworth, J.; Padua-Perez, N.; and Emad, F. 2006. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. In *ITiCSE '06: Proceedings of the 11th annual conference on Innovation and technology in computer science education*. ACM Press.

Spacco, J.; Hovemeyer, D.; and Pugh, W. 2004. An eclipse-based course project snapshot and submission system. In *3rd Eclipse Technology Exchange Workshop (eTX)*.

Winters, T., and Payne, T. 2005. What do students know? In *Proceedings of ICER 05*.

Winters, T.; Shelton, C. R.; Payne, T.; and Mei, G. 2005. Topic extraction from item-level grades. In Beck, J., ed., *American Association for Artificial Intelligence 2005 Workshop on Educational Datamining*.