Department of Computer Science
University of Toronto
http://learning.cs.toronto.edu

6 King's College Rd, Toronto
M5S 3G4, Canada
fax: +1 416 978 1455

Funded in part by NSERC.

**September 13, 2004**

**UTML TR 2004–002**

# CVS Data Extraction and Analysis: A Case Study

**Keir B. Mierle**, **Kevin Laven**, **Sam T. Roweis**, **Greg V. Wilson**
Department of Computer Science, University of Toronto

## Abstract

Version control repositories contain a wealth of detailed information about the evolution of a codebase. In this paper, we outline our experiences parsing and analyzing data from a large collection of CVS repositories created by many students working on a small set of assignments in a second year undergraduate computer science course. We believe the data to be quite unique because rather than a single team working on a single large project we have many individuals working separatly on identical projects. We describe a system for parsing repositories into a SQL database. From the database and repositories we extract various statistical measures of the code and version histories. Finally, we attempt to correlate the code and repository features with grades. Despite using 166 features derived from the code and repository histories, we find that grade performance cannot be accurately predicted; certainly no predictors stronger than simple lines-of-code were found.

# CVS Data Extraction and Analysis: A Case Study

**Keir B. Mierle**, **Kevin Laven**, **Sam T. Roweis**, **Greg V. Wilson**
Department of Computer Science, University of Toronto

## 1 Introduction

Version control repositories contain a wealth of detailed information about the evolution of a codebase. In this paper, we outline our experiences analyzing data from a large collection of CVS repositories created by many students working on a small set of assignments in a second year undergraduate computer science course at the University of Toronto.

We believe our data set is rather unique. It contains hundreds of completely independent repositories, one for each student. Each student is implementing the same thing at the same time. Previous work analyzing logs from version control systems has tended to focus on a single large repository involving many coders working on different parts of the same software project[8, 7].

### 1.1 Goals

The broad goal of our research programme is to mine the data contained in the combination of CVS logs and the actual files modified by coders to find statistical patterns or predictors of performance. In particular, for undergraduates, we are interested in finding early indicators of students who may be struggling in order to point them to helpful resources before deadlines have expired. One very simple way to seek such indicators is to predict low grades (e.g. the bottom third of the class) based on a variety of statistics extracted from a student's CVS data.

### 1.2 CVS Background

CVS, the Concurrent Versions System[1], is a source code management system. It provides a facility for storing all past and present versions of a project's codebase; and also automates many aspects of writing software as a team such as merging changes, propagating updates and flagging conflicts. While CVS is by far the most widely used and heavily tested version control system, it is by no means comprehensive. It lacks certain desirable capabilities such as atomic transactions involving several files, tracking of file metadata, clean handling of deletions and renamings, and more.

A CVS repository consists of two parts: administrative files stored in a central location (known as `CVSROOT`), and RCS files associated with each code or data file in the repository. The RCS files store revision histories for the individual files comprising the project.

This storage scheme is a bit complex to parse; to the best of our knowledge, there are no publicly available libraries for accessing CVS repositories, only the command line tool `cvs`.[1] To further complicate matters, in CVS there are two separate and occasionally disjoint records of activity in a repository: the `CVSROOT/history` file and the individual RCS files (ending in `,v`). File histories are implicitly stored in the `RCS` files which record modifications, additions, and scheduled deletions. The `history` file tracks (almost) all interactions between users and a repository including those listed above as well as checkouts, updates, and conflict resolutions.

[1] An alternate system called `subversion`[5], on the other hand, has a clean API for accessing repository data.

Unfortunately, the history file does not record the initial importing (creation of a local copy) of a project.

Another problem is that of grouping transactions. Unlike many other source code management programs, CVS does not log transactions executed with a single client command but involving multiple files as an atomic event; instead, each file is processed and logged sequentially and receives an individual version number that has no connection with the versions of other files taking part in the same simultaneous transaction.

# 2 Data Extraction and Storage in SQL

Our original code for extracting data from CVS repositories was based[2] on `ViewCVS`[6], which includes a fast RCS parser written in C++.

Once the transaction data have been extracted from the RCS files and parsed into appropriate fields, we load it into a SQL database for convenient storage. SQL was chosen over XML (after some debate) because there are several freely available SQL database engines which allow easy and portable data extraction via SQL dumps, and at the same time permit complex queries. (Our current system uses MySQL[2], although we now believe PostgreSQL[4] to be superior since subselects, transactions, and foreign keys are difficult in MySQL.)

## 2.1 Transaction Clumping

We also used a variant of a simple heuristic[9] to clump groups of transactions into single events. CVS, because of its RCS heritage, doesn't group together the log entries for multiple transactions that are all part of the same single command (e.g. commit); so we have to attempt to re-group them ourselves based on temporal proximity and other details of the transactions. (Of course, the information about the individual transaction is still preserved in our database, the grouping only attempts to reconstruct extra information.)

The sliding window approach outlined in[9] provides the basis for our technique. In this approach, any set of transactions with the same author and comment string, in which neighbouring transactions occur within $\tau$ seconds of each other are grouped into a single event. (This is slightly different from a fixed time window approach, in which all transactions in an event must fall within $\tau$ seconds of the first one in that event.)

In our data, a window of $\tau = 50$ seconds seems to work well. While this differs slightly from the results in[9], this is not unexpected. Our data set consists of many small events, most of them performed locally, and thus taking very little time. Most of the literature contains results from large projects, in which events are larger and less frequent. As a result, "over-clumping" is more of a concern in our data, and a smaller time window is appropriate.

Two modifications were made to the sliding window approach to improve our results. First, it was noted that some CVS clients allow the user to enter a different comment for each file involved in a single event. By adding an additional constraint that all transactions by the same user that occur at the same second be grouped together, we were able to reduce the number of events that were incorrectly split because they contained multiple comment strings. Second, it was noted that a single file cannot have multiple modifications within one event. After the clumping was complete, any event which contained the same file more than once was split into separate events, decreasing the amount of over-clumping.

## 2.2 Database Schema

The database schema we designed is not a general CVS analysis schema; rather it was fairly specific to our data. However, we also created a cleaner version which is totally general and

[2]We modified the code so that it compiled under GCC 3.2 and eliminated many memory leaks.

can import any CVS repository.

In our case, we created two sets of tables. The first set of tables stores information about the coders (in this case students) and projects (in this case course assignments):

**Courses** A list of courses with information about each one, for example start date, end date, course title, instructor, location of students CVS repositories, etc.

**Students** A list of each student by their unix login.

**Enrollment** Which students are enrolled in which course.

**Assignments** A list of each assignment, with the post date, due date, weight, description.

**Grades** A list of login, assignment, grade.

The second set of tables, which are the only tables in the cleaned version of our parser, store information about CVS operations (transactions) executed by the coders and the details of some of those operations (the ones which modify files):

**Transactions** Each CVS operation – for example Modify foo.c or Update bar.h – shows up as a single record in this table. This table is built from the CVSROOT/history file which contains information about every CVS transaction. No RCS files are used when building this table.
Each transaction is tied to a student and a course.

**Revisions** For all transactions where files are actually modified,[3] there is a corresponding RCS revision inside a RCS file. This table tracks revisions, including diff size and log entry (e.g., `Fixed a bug in Parser.java`). Each RCS file starts with an Add followed by any number of modifies.

Revisions are a subset of transactions. Each revision is essentially a transaction with diff text, log entry, and an assignment ID. We use the information in the RCS file to find the matching transaction ID from `transactions`.

Revisions are special because they can be identified as being part of a specific assignment. In the setup used for this particular 2nd year course, each student had one repository for the entire term. Each assignment had its own directory in the repository, for example `csc207/exer/e04`. While parsing revisions, each revision is given an assignment based on the path of the RCS file within the CVS repository.

Our Python program – affectionately called `slurp` – crawls over the repositories and `history` files to populate the database. It parses external files containing class enrollment information, student grades, etc. and also matches transactions from the CVS history files with individual RCS files documenting specific transactions.

## 3 Data Analysis and Visualization

Having parsed the raw data files and loaded the resulting data structures into a structured database, we created various tools to analyze the data both visually and more quantitatively.

Starting out with visualization, we have examined the data using a variety of different displays. These include views which aggregate statistics across all students and projects as well as specialized views which allow us to examine the behaviour of a single student and/or a single assignment (project).

[3] Specifically, of type Add, Remove, or Modify.

In particular, we have examined the relation between final course grade and number of lines of code produced by each student. We can also look at the relationship between the two components contributing to final course grade, namely assignment marks (term mark) and performance on a final exam (not represented in the CVS repository). Examples of such plots are shown in figure 1 below. They show informally that students who write very little code tend to do poorly (but beyond a certain point writing more code does not correlate with higher grades) and that (with a few exceptions) students who do well on assignments also do well on the exam. We have also performed a more quantitative mutual information analysis (see below) showing that the number of lines of code written is a statistically significant (though weak) predictor of grade at the p=.05 level, and is a stronger predictor than any other complex feature we were able to find.

This relationship is not surprising considering that students who do not write enough code to complete an assignment necessarily get low grades. Once a student writes enough code to finish an assignment, lines of code are no longer a strong indicator of quality.

Rather than just looking at total lines of code produced or other features of the code written, we can also look at the repository activity of each student. In this vein, we compared student grades with the number of transactions of various types executed by the student, hoping to see that a certain type of transaction (or mix of types) is particularly indicative or anti-indicative of performance. The displays, however, suggest that no particular transaction types are indicative of high or low performance (see figure 2 below). This is confirmed by our mutual information analysis which shows that none of transaction type counts have statistically significant mutual information with grade.

Beyond counting transactions, we can also analyze their temporal statistics since the CVS logs contain the time at which each one was executed. One simple measure is the average time between two consecutive transactions of the same type, as shown in the histograms of figure 4 below, using a logarithmic axis for time. Surprisingly we see that there are both very small gaps and very long gaps even in a course limited to 13 weeks where each assignment lasts for at most a few weeks.

We have also experimented with visualization of the data at a more specific project level, rather than aggregating over all students. In particular, we have developed a time-series "strip chart" displaying all the transactions executed by a particular student for a single assignment. Such a view allows us to see a complete visualization of all the CVS activity for one project (since projects do not share code and students do not work together on the assignments analyzed here).

### 3.1 Quantitative Statistical Summaries of Repository Information

In addition to visualization of the data, we also applied more quantitative algorithms to relate repository statistics and features of the code files to student grades.

To apply these algorithms, we first had to convert all of the known data about each student into a set of numerical summary statistics (called "features") to be used as predictors for student performance.

Our system analyzes the transaction histories, log comments and details of the actual code files for each student and collects over 150 features. Some of the features are extracted from the database. The rest are taken by checking out each students repositories and crawling them for features. In figure 5 the top ranking features are listed, ranked by their mutual information.

Essentially, there were three classes of features. The first features we calculated from the database of CVS data. No code features were used for these numbers. They include things like the average number of revisions per file, number of local CVS operations, number of update transactions, how long do they wait before the deadline to submit the assignment, and more.

The other part is parsing the Python and Java code directly. For these features, each

students repositories were checked out and crawled. Two simple parsers were written in Python to look for features directly in Python and Java code; for example, number of while loops, number of comments, check for formatting habits, etc.

In addition to our custom parser, we used PMD[3] to examine all Java files. PMD detects many types of higher-level features, particularly style violations or bad practices. It detects things like variable names that don't follow a naming standard, boolean expressions that can be simplified, empty if statements, asserts without a message, and a whole slew of others. The PMD website has more information.

### 3.2 Mutual Information for Feature Evaluation and Induction

The mutual information between two random variables is a measure of their dependence or independence. It is a more stringent measure than the traditional correlation coefficient, which only measures average second order statistics but cannot capture complex higher order dependencies. In particular, the mutual information between two (discrete) random variables $x$ and $y$ is defined as the cross-entropy between their joint distribution and the product of their marginal distributions:

$$I(x,y) = KL[p(x,y)\|p(x)p(y)] = \sum_{x,y} p(x,y)\log_2 \frac{p(x,y)}{p(x)p(y)}$$

For our data, we did not have enough samples to accurately estimate the joint density between all the features we investigated. However, we do have enough data to estimate the mutual information between a single feature and student grades. Specifically, we created a binary random variable $y$ which was 1 if a student achieved a grade which placed them in the top third of the class and zero if it placed them in the bottom third of the class. Students in the middle third were excluded from the estimation procedure. We then discretized each feature $f$ into $K$=20 bins (with ranges $f_k$) and computed the mutual information between this discretized random variable and the binary grade variable as follows:

$$I(f) = \sum_{y=0,1}\sum_{k=1}^{K} p(y)p(f \in f_k|y)\log_2 \frac{p(f \in f_k|y)}{p(f \in f_k)}$$

where $p(f \in f_k)$ is the overall observed frequency with which the feature falls into bin $k$ and $p(f \in f_k|y)$ is the frequency for either the high or the low grade students. $p(y=1) = p(y=0) = 0.5$ since we select exactly equal numbers of students with high grades (the top third of the class) and low grades (the bottom third of the class).

Using these estimates, we can rank the features by how much information they contain about the course grade. Figure 5 gives a short list of the top features and their estimated mutual information .The result was quite surprising to us:

*Of all the 166 features we examined (including the ones listed above) we found that total lines of code had the highest mutual information with course grade; furthermore it was one of only 3 features to have significant dependence on grade.*

Given the number of student repositories used, only three features had statistically significant mutual information with the grade of the student at the p=0.05 level: lines of code written by the student, number of commas followed by spaces[4] and total length of diff text[5] (In this case $I = 0.22$ bits was the significance cutoff.) These are indicated with (*) in figure 5.

[4]Consider `foo(a,b,c,d)` instead of `foo(a, b, c, d)`; the latter shows more care taken to format the code.

[5]Each time a student does `cvs commit`, only changes to the code are stored. The "total diff length" feature is the total character count of all the deltas combined.

### 3.3 A Machine Learning Approach To Grade Prediction

With numeric features in hand, summarizing the repository data for each student, we were in a position to try a variety of statistical pattern recognition algorithms for predicting grades based on the features. Of course, the lack of significant mutual information between grades and most of the features we extracted do not bode well for such an enterprise, but we have conducted several experiments nonetheless and report their results here.

We used three very basic algorithms from the machine learning and applied statistics fields: nearest neighbour classification, Naive Bayes and logistic regression. Before we could classify, we normalized the features to have zero mean and unit variance. (We applied this normalization to all features even those whose histograms were obviously not Gaussian.)

As expected, none of the classifiers were able to reliably predict grades for students based on the features given. While some algorithms managed to overfit the training sets and achieve 0% training error, the errors on an independently held out test set were always far inferior. A leave-one-out (LOO) cross validation estimate of test error was typically around 25%. In particular, for logistic regression, our LOO error was 29.7%, for Naive Bayes it was 23.9% (using discretized versions of the features), and for nearest neighbour it was also 23.9% (at K=21 using Euclidean distance in the normalized feature space). In all these experiments, we used 166 normalized features to classify 69 students from the top third of the class (by grade) from 69 students from the bottom third.

## 4 Conclusion

We have described the results of analyzing data from a large collection of CVS repositories created by many coders (students) working on a small set of identical projects (course assignments). We have implemented a complete system for parsing such repositories into a SQL database and for extracting, from the database and repositories, various statistical measures of the code and version histories.

Although version control repositories contain a wealth of detailed information both in the transaction histories and in the actual files modified by the users, we were unable to find any measurements in the hundreds we examined which accurately predicted student performance as measured by final course grades; certainly no predictor stronger than simple lines-of-code-written was found.

These results seem to challenge the conventional wisdom that a repository contains easily extractable predictive information about external performance measures. We are eager to have other researchers suggest novel measures which, contrary to our efforts, contain substantial information about productivity, grades, or performance.

### Acknowledgments

We thank Karen Reid, Michelle Craig, Kevin Laven and Eleni Stroulia for helpful discussions about the data and analysis tools. STR is supported in part by the Canada Research Chairs program and by the IRIS program of NCE.

## References

[1] CVS. `http://www.cvs.org/`.

[2] MySQL. `http://www.mysql.com/`.

[3] PMD: A style checker. `http://pmd.sourceforge.net/`.

[4] PostgreSQL. http://www.postgresql.org/.

[5] Subversion: A compelling replacement for CVS. http://subversion.tigris.org/.

[6] ViewCVS. http://viewcvs.sourceforge.net/.

[7] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. In *IEEE Transactions on Software Engineering*, volume 26, July 2000.

[8] Ying Liu, Eleni Stroulia, Kenny Wong, and Danial German. Using CVS historical information to understand how students develop software. In *Proc. International Workshop on Mining Software Repositories (MSR04), Edinburgh*, 2004.

[9] Thomas Zimmermann and Peter Weibgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR04), Edinburgh*, 2004.

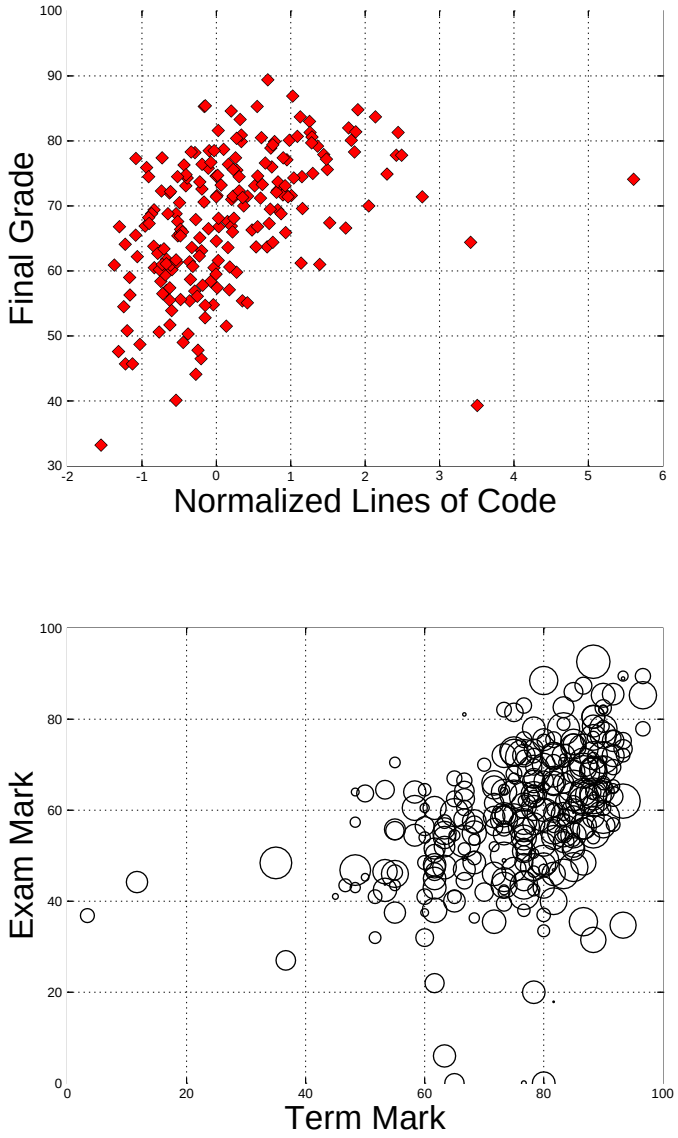


Figure 1: **(top)** Final grade versus normalized lines of code the student wrote. (Normalization was done by subtracting the mean and dividing by the standard deviation.) Each diamond represents a single student. The horizontal position of the diamond represents the number of lines of code a student wrote over the term. The vertical position represents their final grade in the course. The correlation visible in this graph between grade and lines of code is as strong as the correlation between grade and any other complex feature we were able to find.**(bottom)** The relationship between the two components making up a final grade. Horizontal position shows the marks on the coding assignments ("term mark"); vertical position shows performance on a final exam. Radius of each circle indicates the total number of CVS transactions executed by the student.

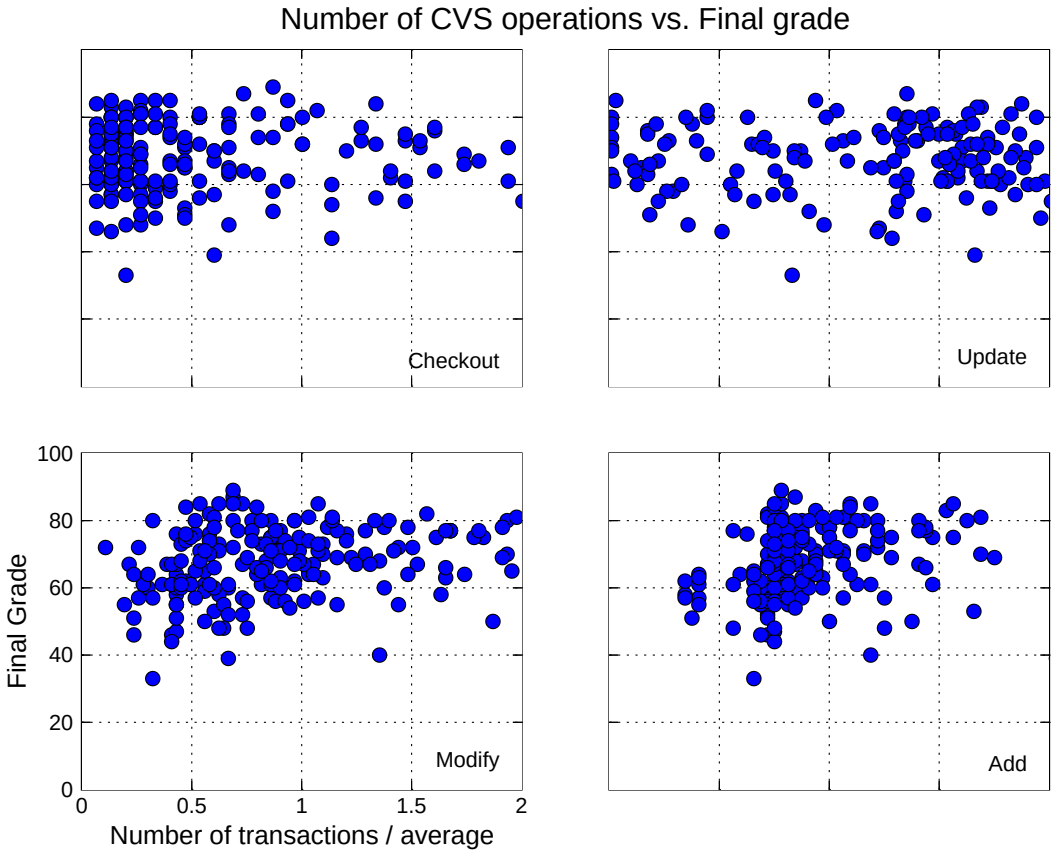


Figure 2: Final grade versus normalized frequency of transactions of various types. (Normalization was done by dividing out the mean.) Each circle represents a single student. The vertical position of the circle in each panel gives the student's final grade in the course while the horizontal position represents a normalized number of transactions of a particular type. Visually, there is no strong correlation present between any of these frequencies and grades; this is confirmed quantitatively by the mutual information analysis in figure 5.

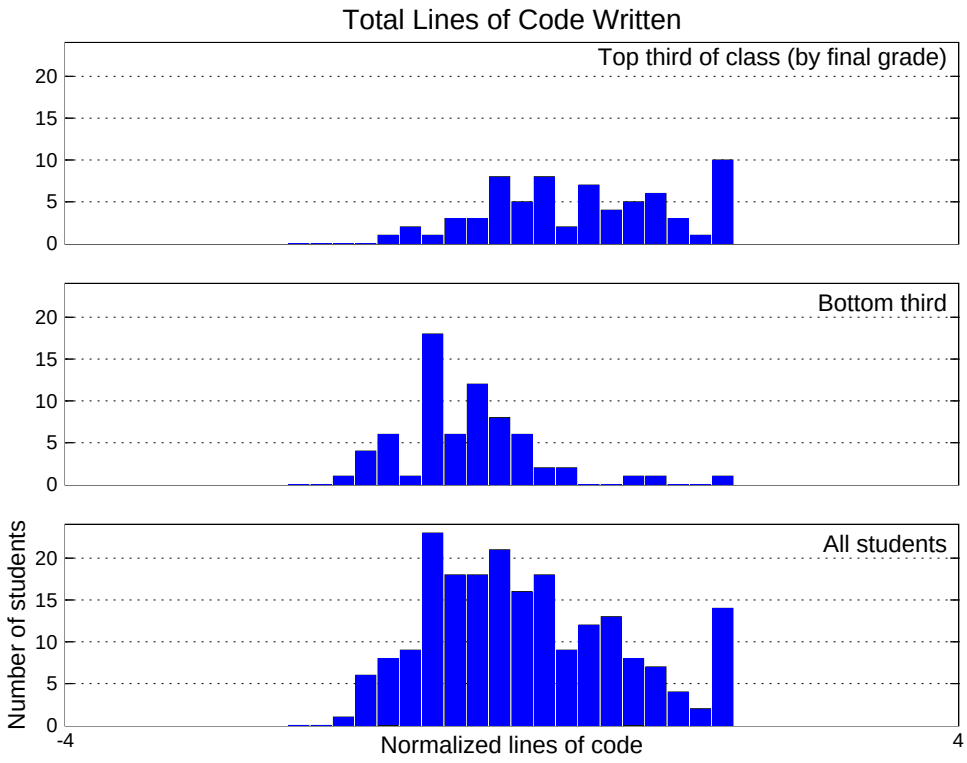


Figure 3: Histograms of (normalized) lines of code written, for the top third (by grade), bottom third, and entire class of 207 students. Visually, it can be seen that students who write more code are more likely to be achieve high grades. This feature was the top scoring predictor of grade in our mutual information analysis and is a statistically significant (though weak) predictor of high grade at the p=.05 level.

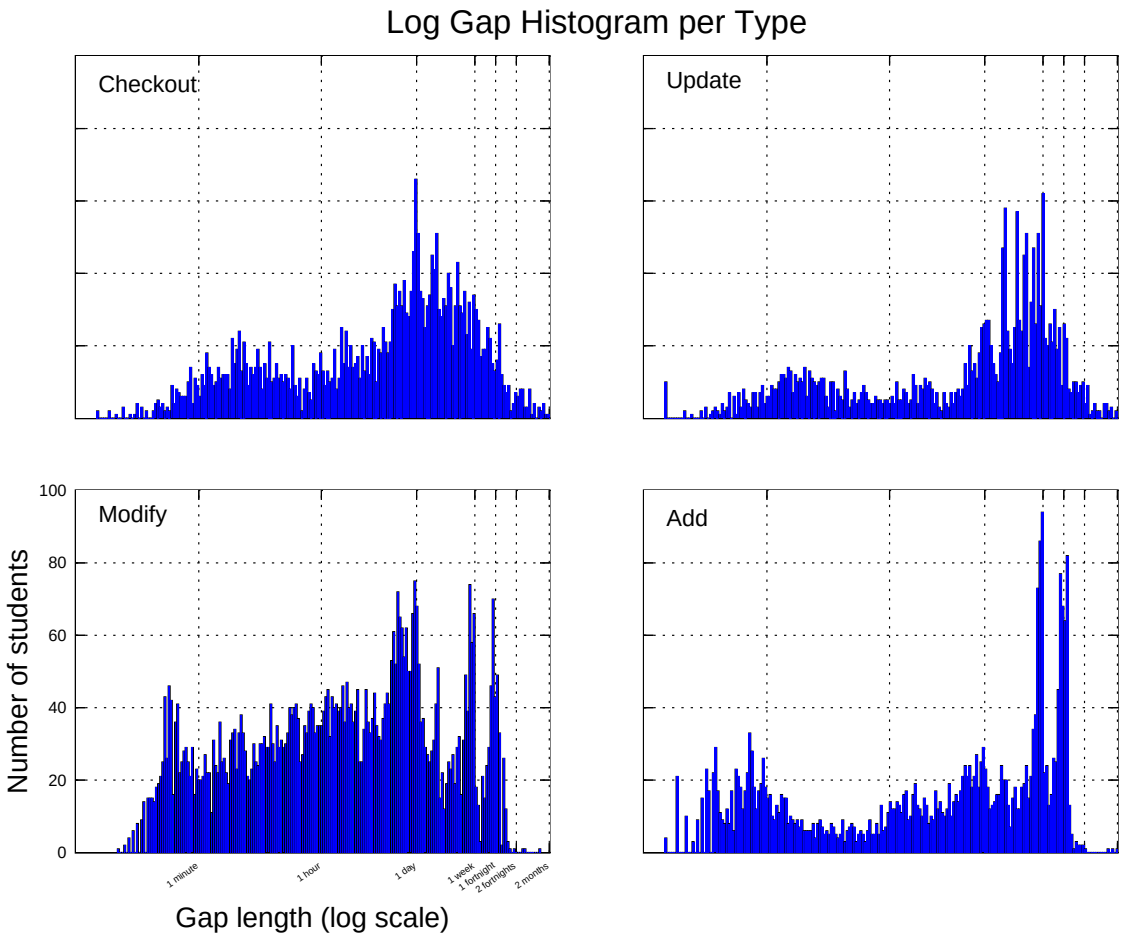


Figure 4: Histograms of the time between consecutive transactions of the same type. The (horizontal) temporal axis is spaced logarithmically showing that there are both very small gaps and very long gaps even in a course limited to 13 weeks where each assignment lasts for at most a few weeks.

| Mutual Information (bits) | Feature Description |
|---|---|
| 0.29 | (*)# newline characters in files the student added or modified |
| 0.28 | (*)# times a space followed a comma, e.g. "`foo(a, b, c)`" |
| 0.26 | (*)# characters in diff text between successive revisions (CVS) |
| 0.20 | # comments (Python) |
| 0.20 | # literal strings (Python) |
| 0.19 | # operators (Python) |
| 0.16 | # characters in all comments |
| 0.16 | # function/method definitions (Python) |
| 0.14 | # `while` loops (Python) |
| 0.14 | # Terminal tokens (Python) |
| 0.13 | # 4-space indents |
| 0.13 | # comment-space-capital sequences e.g. "`// Nicely formatted`" |
| 0.12 | # commits (CVS) |
| 0.12 | # `for` loops (Python) |
| 0.11 | # newlines (Python) |
| 0.11 | # files in repository |
| 0.11 | # violations of "Assertions should include a message" (PMD) |
| 0.11 | # `self` references (Python) |
| 0.10 | # modifies (CVS) |
| 0.10 | # violations of "Avoid duplicate literals" (PMD) |
| 0.10 | # `except` tokens (Python) |
| 0.10 | # leading tabs |
| 0.10 | # total transactions (CVS) |
| 0.09 | Average revisions per file (CVS) |

Figure 5: Mutual information between various features of a student's repository and the binary indicator of whether they fall into the top third or bottom third of the class (by final grade). The mutual information is a measure of how related two variables are. Here we calculated the mutual information between the features and the binary grade for the top third and bottom third of students only. A partial list of features is shown, sorted by mutual information with grade. In total, 166 features from 207 student repositories were used. Features marked (Python) were measured by crawling all `.py` files in the student's repository. Likewise, the (CVS) marking means the feature was measured from the CVS logs, and (PMD) denotes how many rule violations PMD found. Only values greater than 0.22 bits are statistically significant at the p=.05 level, given the number of students. Significant features are indicated with (*).