Statistical Debugging: Simultaneous Identification of Multiple Bugs

Alice X. Zheng ALICEZ@CS.CMU.EDU
Carnegie Mellon University, School of Computer Science, Pittsburgh, PA

Michael I. Jordan JORDAN@CS.BERKELEY.EDU
University of California, Berkeley, Department of EECS, Department of Statistics, Berkeley, CA

Ben Liblit LIBLIT@CS.WISC.EDU
Computer Sciences Department, University of Wisconsin-Madison, Madison, WI

Mayur Naik MHN@CS.STANFORD.EDU
Alex Aiken AIKEN@CS.STANFORD.EDU
Computer Science Department, Stanford University, Stanford, CA

Abstract

We describe a statistical approach to software debugging in the presence of multiple bugs. Due to sparse sampling issues and complex interaction between program predicates, many generic off-the-shelf algorithms fail to select useful bug predictors. Taking inspiration from bi-clustering algorithms, we propose an iterative collective voting scheme for the program runs and predicates. We demonstrate successful debugging results on several real world programs and a large debugging benchmark suite.

1. Introduction

Traditional software debugging is an arduous task that requires time, effort, and a good understanding of the source code. Given the scale and complexity of the task, the development of methods for automatically debugging software seems both essential and very difficult. However, several trends make such an endeavor increasingly realistic: (1) the wide-scale deployment of software, (2) the establishment of distributed crash report feedback systems, and (3) the development of statistical machine learning algorithms that can take advantage of aggregate data over multiple users.

The statistical approach to software debugging that we pursue here is based on a fine-grained instrumentation of software and a distributed data collection infrastructure (Liblit et al., 2003). The process starts with a source-to-source transformation of a (presumably) buggy C program. This transformation creates a large number of instrumented sites that assess the truth value of predicates over the run-time state of program variables. Each time a given instrumentation site is reached during program execution, the instrumentation code at the site is optionally executed, according to a probability distribution that can be tuned to minimize the instrumentation's impact on run-time performance. If the code at a site is executed, we record the true/false value of the corresponding predicate and say that the predicate is *observed*. Otherwise, the program skips the site and moves on (and the predicate is *unobserved*). Feedback reports contain the counts of the number of times each predicate is found to be true during each run of the program. In addition, runs are labeled as either successful or failing, depending on the exit status of the program.

Since the instrumentation predicates are automatically generated, most of them do not provide any useful information about the locations of bugs. The primary problem of statistical debugging is thus to select the most useful bug-predicting predicates from a set of user feedback reports. The sheer size of the data sets involved can be problematic for some naïve approaches. One of our test programs, for example, contains 56K lines of code and 857K predicates, out of which only a handful of predicates are useful bug predictors. Large software programs contain millions of lines of code and hundreds of millions of candidate predicates for an algorithm to choose from.

Appearing in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA, 2006.

Sparse sampling of predicates and the complexity of their interactions further complicate the matter. In the simple case of programs containing a single bug, the automatic debugging problem can be posed as a feature selection problem in the binary classification context. Relatively straightforward, off-the-shelf techniques work well in this case (Zheng et al., 2004). However, we have found that these algorithms fail when applied to more realistic cases of programs containing multiple bugs, because the algorithms have difficulty coping with issues related to sampling and cannot distinguish between useful bug predictors and predicates that are secondary manifestations of bugs.

In this paper, we attack the multi-bug problem. We first identify some of the major challenges arising in the multi-bug case that cause simple solutions to fail in Section 2. We design a new algorithm that addresses these challenges in Section 3. In Section 4, we present results from experiments on real programs in which our algorithm is able to identify known and previously unknown bugs. We also compare our algorithm with two other statistical debugging techniques on a suite of 130 test programs. We conclude in Section 5.

2. Multi-Bug Challenges

In the multi-bug setting, we assume that a program contains multiple bugs. Failing runs are of course not labeled according to the bug that causes failure, and thus the problem has somewhat of the flavor of a clustering problem. There are some idiosyncrasies, however. Note that we have both failing runs and successful runs, so the problem retains an aspect of classification. Moreover, any given run can exhibit more than one bug. Finally, and most importantly, our problem is not simply identifying clusters, but finding features that allow us to identify clusters and simultaneously characterize failing versus successful runs. Thus the problem is an instance of a feature selection problem, but one that does not appear to have an off-the-shelf solution.

In order to make these issues concrete, we briefly present the results of applying several simple algorithms to a realistic multi-bug problem. We do not claim that the algorithms that we use here are in any sense an effective choice; we use them simply to highlight the issues that arise in the multi-bug setting.

Our testbed is MOSS, a software plagiarism detection program with a large user community. We introduce variations of nine bugs found in previous versions of the software and generate runs of the program by varying command-line options and input files. As it turns out, bug #8 never manages to trigger any failed runs and is thus ignored hereafter. Bug #1 causes incorrect output; the rest of the bugs crash the program in various ways. We tag the failed runs by their exit error signal, and compare their output against results from a version of MOSS without bugs. Note also that in these experiments (and in all subsequent experiments in this paper), we pre-filter the data according to a statistical test that eliminates the bulk of the uninteresting predicates. In particular, we retain predicate i only if it passes the following simple test:

$$P(\text{pred } i \text{ is true and run fails} \mid \text{pred } i \text{ observed}) > P(\text{pred } i \text{ is true and run succeeds} \mid \text{pred } i \text{ observed}),$$

where P is estimated via empirical counts.

Table 1: Result of applying the single-bug algorithm to the MOSS data.

Predicate
`(p + passage_index)->last_line < 4`
`(p + passage_index)->first_line < i`
`i > 20`
`i > 26`
`(p + passage_index)->last_line < i`
`i > 23`
`(p + passage_index)->last_line == next`
`i > 22`
`i > 25`
`i > 28`

2.1. A Single-Bug Algorithm

We first examine the results of running the single-bug algorithm of Zheng et al. (2004) on the MOSS dataset. This is essentially a classification algorithm with feature selection via L1 penalty. Table 1 presents the top ten predicates selected by this algorithm. The predicates fall into two groups; none of the predicates is very useful for bug-finding. The (`i > ...`) predicates simply count the length of the command-line. In our experiments, longer command-lines are correlated with failed runs, though successful runs often have long command-lines as well. Hence, while these predicates cover many failed runs, they are not strong bug predictors overall.

The other group of predicates are also of limited usefulness, but for a different reason. Specifically, the (`(p+passage_index)-> ...`) predicates are specific conditions that are satisfied only in certain instances of bug #9. These predicates may sometimes be highly indicative of sub-modes of the bug, but do not cover all of the runs that crash due to bug #9.

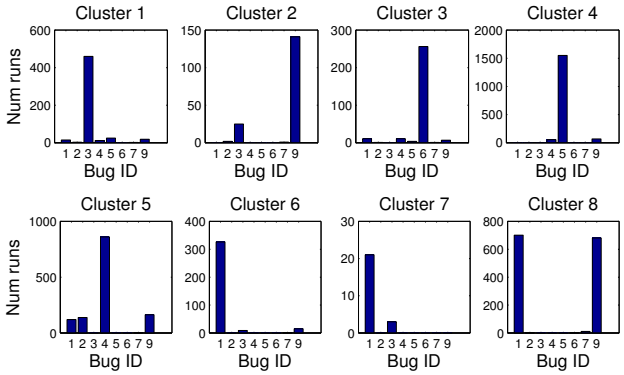

Figure 1: Bug histograms of MOSS run clusters returned by K-means.

In addition to these issues which illustrate that the single-bug algorithm can fail in terms of both sensitivity and specificity, it is also worth noting the high degree of redundancy in the list of the top ten predicates. A more effective algorithm would return a single highly-predictive predicate for each bug.

2.2. Clustering Runs

Let us leave behind the classification perspective, and attempt to treat the multi-bug problem using a standard clustering tool. In particular, we apply K-means clustering on the failed MOSS runs in hopes of resolving the underlying true bug labels. (K-means clustering yielded similar results on two other test programs, RHYTHMBOX and EXIF.)

Each run is represented by a vector of (non-binary) predicate counts. We include only predicates with nonzero variance, and center and normalize each predicate dimension by subtracting the sample mean and dividing by the sample standard deviation.

To make the problem particularly easy for the algorithm, we set the number of clusters equal to the true number of bugs in MOSS, thus $K = 8$. We repeat K-means several times with random initialization and pick the clustering with the smallest intra-cluster distance. Figure 1 shows the resulting "bug histograms"—a representation in which each bin contains the number of runs failing due to that bug. An ideal bug histogram would exhibit peaks at distinct bugs. Clusters 1, 3 and 4 clearly capture bugs #3, #6, and #5, respectively. The rest of the clusters are much less distinct. Some clusters contain multiple bugs, while others contain subsets of a single bug. In particular, bugs #1 and #9 are scattered across multiple K-means clusters.

A closer analysis of the runs in each cluster reveals why naïve clustering fails. The clusters are capturing usage modes of the program, not failure modes. In hindsight, this is not surprising—the usage modes are much more salient statistically than the failure modes. Without additional constraints, clustering algorithms home in on program usage modes, which may provide little leverage for bug detection.

2.3. Clustering Predicates

As we saw in our attempt to use the single-bug algorithm (and as we have seen when using other algorithms), predicate redundancy can be a significant problem. Users of a statistical debugging tool do not want to wade through a long list of redundant predicates, particularly if the redundancy is not obvious. To address this problem, and to begin to address issues of feature selection in the clustering context, we can attempt to cluster predicates.

In studying this problem, we found that it interacts strongly with the sparsity that is characteristic of our domain. Suppose predicates a, b, and c are mutually redundant. At a sampling rate of d, it takes $O(1/d^2)$ runs for each predicate pair to be co-observed, and $O(1/d^3)$ runs for all three to be observed together. In practice, there may not be enough runs in the dataset for us to observe large values of similarity among all redundant predicates. It is much more probable that we would observe, for instance, that a is close to b, and b is close to c, but a is not close to c.

These considerations led us to explore spectral clustering methods for predicate clustering, given the ability of the spectral approach to respect transitivity. Spectral clustering requires a similarity metric, which we took to be the product-moment correlation coefficient between pairs of predicates. In order to reduce bias arising from the predicate sampling process, we must condition on the observation status of predicates. Assuming that predicates from different instrumentation sites are independent of each other, the conditional correlation coefficient can be written as:

$$\rho(X, Y \mid X, Y \text{ observed}) = \frac{p_{xy} - p_x p_y}{\sqrt{p_x(1-p_x)p_y(1-p_y)}},$$

where

$$\begin{aligned} p_x &:= P(X = 1 \mid X \text{ observed}), \\ p_y &:= P(Y = 1 \mid Y \text{ observed}), \\ p_{xy} &:= P(X = 1, Y = 1 \mid X, Y \text{ observed}). \end{aligned}$$

Using the spectral clustering algorithm of Ng et al. (2002), we vary the number of clusters from 5 to 20 and pick the clustering with the smallest average intra-cluster distance (the result turned out to be 9).

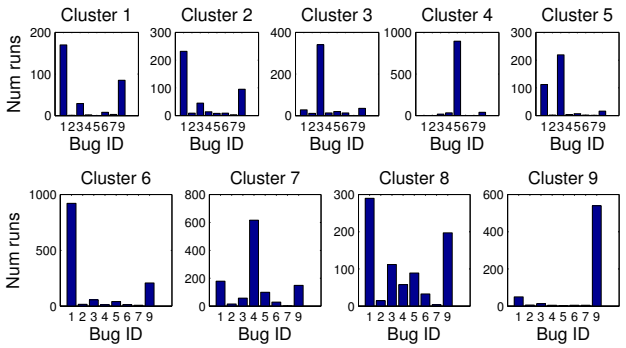

Figure 2: Average bug histograms of MOSS predicate clusters.

Figure 2 plots the average bug histograms of the predicate clusters. For each cluster, we count the number of runs exhibiting each bug, averaged over the number of predicates in that cluster. Most of the bug histograms in Figure 2 do not contain a single predominant peak. The only exception is cluster 4, which contains most of the predictors for bug #5. All other clusters contain mixtures or subsets of bugs.

These non-distinct bug histograms arise due to the presence of what we refer to as *super-bug predictors*. Super-bug predictors are usually very general pre-conditions for failure — the predicates measuring command-line length in Table 1 are example super-bug predictors. Super-bug predictors loosely correlate with many failed runs but also tend to be true in some number of successful runs. The MOSS dataset contains some prominent super-bug predictors for bugs #1 and #9, which bond the predicate clusters together. The resulting super-cluster is then broken along weaker links, leaving a set of scrambled predicates that do not correspond neatly to bugs.

2.4. Issues

We have identified several issues that any successful multi-bug algorithm must face. First, some predicates are non-specific *super-bug predictors* that make feature selection difficult; they are useful in multiple clusters and thus are highly supported. Second, other predicates are *sub-bug predictors*, highly-specific predictors that home in on specific aspects of an individual bug but fail to reveal the general case. Third, *predicate redundancy* is a problem, both algorithmically and in terms of feedback to the user. Finally, interlaced with all of these issues is the issue of *sampling*. The data collection framework samples predicate so that the instrumentation does not impact the performance of a running program. But this creates subtle statistical biases and linkages that complicate the core issues of clustering and feature selection.

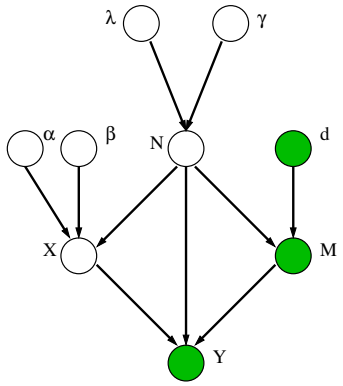

Figure 3: Predicate truth probability model.

3. Algorithm

The algorithm that we propose in this section has two parts. The first part deals with sampled predicates using a graphical model to infer truth probabilities from observed data. The inferred probabilities serve as input to the second stage, which is a bi-clustering algorithm that attempts to select features by jointly clustering runs and predicates.

3.1. Inferring Truth Probabilities

Based on the observed counts and what we know about the sampling process, we can make probability statements about the truth of each of the predicates in any particular run. In doing so, we hope to ameliorate the data sparsity problem, reducing bias caused by sampling and thereby isolating these issues from those associated with clustering and feature selection.

Figure 3 is a graphical model for the truth probability of a single predicate. M is a random variable representing the number of times a predicate is observed in a particular run of the program. Y denotes the number of times it is observed to be true. X is the actual number of times that the predicate was true in that run, and N is the number of times the instrumentation site was reached. We observe M and Y, but not N and X. Our ultimate goal is to compute the posterior truth probability $P(X > 0 \mid M, Y)$ for each predicate in each run.

In the MOSS dataset, we observe that the number of times a site is reached often follows a Poisson-like distribution. However, during certain runs, the site may not be reached at all, leading to an additional spike at zero. Therefore we endow N with a prior that is a mixture of a Poisson distribution and a spike at zero. Given N, we model X as a mixture of a binomial distribution, a spike at zero, and a spike at N. The spikes represent "sticky" modes where the predicate

is either never true or always true. Lastly, based on the predicate sampling process, we know that $P(M \mid N)$ is a binomial distribution with parameter d (the sampling rate). Y has a hypergeometric distribution given M, N, and X.

Thus the conditional probabilities of the predicate truth probability model are:

$$\begin{aligned} N &\sim \gamma\text{Poi}(\lambda) + (1-\gamma)\delta(0) \\ X \mid N &\sim \beta_1\text{Bin}(\alpha, N) + \beta_2\delta(0) + \beta_3\delta(N) \\ M \mid N &\sim \text{Bin}(d, N) \\ Y \mid M, N, X &\sim \text{Hypergeo}(M, N, X), \end{aligned}$$

where $\delta(c)$ denotes a delta function at c.

We endow the parameters α, $\boldsymbol{\beta}$, λ, and γ with conjugate priors. Thus, α and γ are beta-distributed, $\boldsymbol{\beta}$ has a Dirichlet prior distribution, and λ has a Gamma distribution. We use an empirical Bayes approach to set all of the hyperparameters given data.

The ultimate goal is to compute the posterior truth probability $P(X > 0 \mid m, y)$. The case where $y > 0$ is trivial since $P(X > 0 \mid M = m, Y > 0) = 1$. Hence we only need to examine the case where $Y = 0$. It is easier to first compute $p_0 := P(X = 0 \mid M = m, Y = 0)$, and then calculate $P(X > 0 \mid M = m, Y = 0) = 1 - p_0$. A short calculation yields the following posteriors:

$$p_0 = \frac{\hat{\beta}_1(1-\hat{\alpha})^m e^{-\hat{\lambda}\hat{\alpha}(1-d)} + \hat{\beta}_2}{\hat{\beta}_1(1-\hat{\alpha})^m + \hat{\beta}_2},$$

for $m > 0$, and for $m = 0$,

$$p_0 = \frac{(\hat{\beta}_1 e^{-\hat{\lambda}\hat{\alpha}(1-d)} + \hat{\beta}_2 + \hat{\beta}_3 e^{-\hat{\lambda}(1-d)})\hat{\gamma}e^{-\hat{\lambda}d} + (1-\hat{\gamma})}{\hat{\gamma}e^{-\hat{\lambda}d} + (1-\hat{\gamma})}.$$

3.2. Collective Voting

Our approach to solving the multi-bug problem reposes on a symmetry principle: *Predicates should group by the runs that they predict; runs should group by the predicates that predict them.* We approach the problem from the perspective of bi-clustering (see, e.g., Dhillon, 2001; Hartigan, 1972). But generic bi-clustering algorithms do not directly address the idiosyncrasies of the predicate-run relationship; the standard distance metrics and information-theoretic objectives are not obviously applicable. We develop a novel instance of a bi-clustering algorithm that encapsulates the specific setting of the statistical debugging problem.

The algorithm essentially performs an iterative collective voting process, alternating between updates of predicate quality and the vote distribution of runs. Each failed run ultimately casts a vote for its favorite predicate. The predicates are then ranked by the number of votes they receive.

In our recursive voting procedure, program predicates are the candidates, and runs are the constituents. A predicate has a "quality" Q_i based on the votes it receives from the runs. Each run has one unit vote to cast. In the beginning, a run distributes fractional votes among the candidate predicates, and a candidate receives a vote proportional to its quality. Predicates must compete for a run's attention, and therefore the more predictors a run has, the smaller vote each predicate receives. The competition between predicates encapsulates the problem of redundancy. The voting process iterates until convergence. At that point, each run must firm its resolve and cast the entire vote for a single predicate. Finally, the predicates are ranked by their votes.

The inputs to the algorithm are the posterior truth probabilities inferred via the procedure described in the previous section. We carry out this calculation for each run and each predicate, and let A_{ij} denote the probability that predicate i is true in run j.

Let $\mathcal{F}$ denote the set of failed runs and $\mathcal{S}$ the set of successful runs. Q_i and $Q_{\bar{i}}$ respectively denote the quality of predicate i and its complement.[1] F_i and S_i respectively measure the contribution of predicate i to $\mathcal{F}$ and $\mathcal{S}$; $F_{\bar{i}}$ and $S_{\bar{i}}$ respectively measure the contribution of the complement of predicate i to $\mathcal{F}$ and $\mathcal{S}$. We define the following set of coupled update equations:

$$Q_i = \frac{F_i}{S_i} \cdot \frac{S_{\bar{i}}}{F_{\bar{i}}}, \quad Q_{\bar{i}} = \frac{1}{Q_i}, \tag{1}$$

$$F_i = \sum_{j \in \mathcal{F}} A_{ij} \frac{R_{ij}}{\sum_k R_{kj}}, \quad F_{\bar{i}} = \sum_{j \in \mathcal{F}} A_{\bar{i}j} \frac{R_{\bar{i}j}}{\sum_{\bar{k}} R_{\bar{k}j}}, \tag{2}$$

$$S_i = \sum_{j \in \mathcal{S}} A_{ij} \frac{R_{ij}}{\sum_k R_{kj}}, \quad S_{\bar{i}} = \sum_{j \in \mathcal{S}} A_{\bar{i}j} \frac{R_{\bar{i}j}}{\sum_{\bar{k}} R_{\bar{k}j}}, \tag{3}$$

$$R_{ij} = \begin{cases} A_{ij}Q_i, \text{ if } j \in \mathcal{F} \\ A_{ij}/Q_i, \text{ if } j \in \mathcal{S} \end{cases} \qquad R_{\bar{i}j} = \begin{cases} A_{\bar{i}j}Q_{\bar{i}}, \text{ if } j \in \mathcal{F} \\ A_{\bar{i}j}/Q_{\bar{i}}. \text{ if } j \in \mathcal{S} \end{cases} \tag{4}$$

Let us take a moment to decipher these equations. Equation 1 dictates that a predicate has high quality if it contributes to failed runs but not successful runs, and if its complement contributes to successful runs but not failed runs. The contribution of predi-

[1] For example, if predicate i is (`f == NULL`), then its complement is the predicate (`f != NULL`). A predicate and its complement may be both true (at different times) during the lifetime of a program run.

Table 2: Summary statistics for the datasets: lines of code, numbers of successes and failures, number of predicates, and the number of top predicates accounting for over 90% of all failures after the voting process.

		Runs			
	# Lines	S	F	# Preds	90%
MOSS	6001	26,239	5505	202,998	14
RHYTHMBOX	56,484	12,530	19,431	857,384	6
EXIF	10,588	30,789	2211	156,476	2

cate i to run j is defined to be A_{ij}, the probability that it is true in that run multiplied by R_{ij}, the vote that run j casts for predicate i, and normalized by the total number of votes cast by run j. (See Equation 2 and Equation 3.) Run j decides how much vote to cast toward predicate i based on Q_i, the predicate's failure prediction strength, and the truth probability A_{ij}. (See Equation 4.)

Let us consider a small test example. Suppose the data set includes one failed run, one successful run, two predicates a and b, and their complements $\bar{a}$ and $\bar{b}$. Predicate a is a good bug predictor that is true only in the failed run, and its complement is only true in the successful run. Predicate b is a generic non-informative super-bug predictor: b and $\bar{b}$ are both true in both runs. Initially, all predicates have equal quality: $Q_a = Q_{\bar{a}} = Q_b = Q_{\bar{b}} = 1$. After the first round of updates, the failed run effectively splits its vote evenly between a and b, and the successful run gives all its vote to b. On the other hand, $\bar{b}$ receives all the vote from the failed run, and $\bar{a}$ and $\bar{b}$ share the vote of the successful run. The updated quality scores are $Q_a = (0.5/0)(0.5/0) = \text{inf}$ and $Q_b = (0.5/1)(0.5/1) = 0.25$. At the end of each round of updates, the quality scores are renormalized to have unit sum. For computation purposes we smooth zero scores with a small additive constant. So essentially Q_a becomes a very large number and Q_b relatively much smaller.

4. Results

4.1. MOSS

Table 2 presents statistics of three programs used in our experiments. For MOSS, the pre-filter reduces the number of predicates from 202,998 to 2645, and our voting algorithm selects 14 predicates to account for over 90% of all failures.

Figure 4 contains bug histograms of the top nine predicates. The numbers of runs attributed to each predicate are included in parentheses at the top of each plot. The bug histograms are much more homogeneous

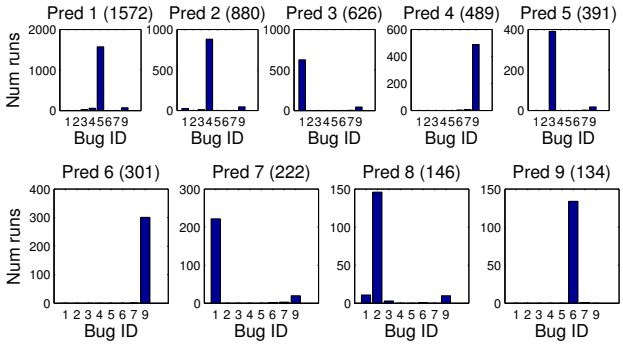

Figure 4: Bug histograms of top MOSS predictors.

Table 3: Top MOSS failure predictors.

Rank	Predicate
1	`files[filesindex].language > 16`
2	`config.tile_size > 500`
3	`config.match_comment is TRUE (line 1994)`
4	`__lengthofp == 200`
5	`i > 52`
6	`i___0 > 500`
7	`config.match_comment is TRUE (line 1993)`
8	`f < yyout`
9	`i >= 8`

than our previous attempts in Section 2. Each predicate (see Table 3) focuses on one specific bug and each bug is represented by at least one predicate.[2] Some bugs are represented by a couple of prominent sub-bug predictors. For instance, predicates 3 and 7 both account for bug #1, which occurs only when C comment-matching is turned on. Predicates 4 and 6 check for the size of a certain array; both predicates account for bug #9, an array-overrun bug. While these sub-bug predictors do not account for the entire suite of failed runs resulting from one bug, they are nevertheless useful indicators of the causes of failure.

We manually verified the quality of each of the top predicates in Table 3. The first eight predicates are all direct indicators of various bugs. But let us focus on the ninth predicate. This predicate turns out to be a super-bug predictor; it is true when the command-line length is longer than 8. Why is a super-bug predictor selected to account for bug #6? The reason turns out to lie in the definition of predicate quality Q_i (Equation 1). Q_i gives a high score to a predicate if it contributes mostly to failed runs and its complement contributes mostly to successful runs. This criterion works well for most bug predictors. However, the only prominent predictor for bug #6 in MOSS involves a command-line flag setting the "`-p`" option. However, the complement of this predicate is also true

[2]Bug #7 is an exception: in our MOSS dataset, it does not have any strong predictors that occur independently of all the other bugs.

in many failed runs, because other command-line options may trigger other bugs. In this case, the prediction strengths of the predicate and its complement are working against each other.

All is not lost. Even though the algorithm fails to select the most informative predictor for bug #6, it nevertheless succeeds in clustering the runs failing due to the same bug. Given the run clusters, we can apply simple single-bug algorithms to select useful bug predictors for each cluster. This has the added advantage of giving us predicate clusters. For example, running a simple univariate hypothesis test [3] on the predicate 9 run cluster yields a predicate "cluster" with these top two predicates:

```
1. STREQ(argv[i], "-p") is TRUE in handle_options();
2. strcmp(argv[i], "-p") == 0 in handle_options().
```

These predicates are equivalent and both indicate that the "-p" option is set. The rest of the predicates on the list are super-bug predictors similar to the ones we have already seen and are not shown.

4.2. RHYTHMBOX and EXIF

We tested our algorithm on two other real-world programs containing multiple bugs. RHYTHMBOX is a graphical, open-source music player that contains at least two bugs. One of the bugs exposes a bad coding pattern related to previously freed and reclaimed event objects. Our discovery of this bug subsequently led to the discovery of other bugs within RHYTHMBOX resulting from the same bad coding practice. (Liblit et al., 2005)

Our algorithm selects the following predicates for RHYTHMBOX.

```
1. (mp->priv)->timer is NULL
2. monkey_media_player_get_uri() == 0
3. vol <= (float) 0 is TRUE
4. (db->priv)->thread_reaper_id >= 12
5. rorder (new val) < rorder (old val)
6. (mp->priv)->tick_timeout_id > 12
```

The first predicate is an important clue for a bug involving a dangling ID of an object that has already been destroyed. The predicate indicates that a pointer to a timer object is NULL, meaning that the object has already been deallocated. However, other parts of the program still holds an ID to this object; subsequent references to the deallocated object cause the program to crash. The second predicate is a predictor of a race condition bug; it indicates that the `monkey_media_player` object has already been destroyed and subsequent callbacks to the object cause a crash. The remaining predicates are manifestations of other unknown bugs in RHYTHMBOX.

EXIF, an open-source image manipulation tool, contains three bugs which are covered by four predicates selected by our algorithm.

```
1. i < k
2. sizeof(JPEGSection)*(data->count-2) < 0
3. machine_readable is TRUE
4. (data->ifd[4])->count is FALSE
```

Predicate clusters of predicates 1 and 3 indicate that they predict the same bug: in EXIF's machine-readable output mode, a function call returns a `NULL` pointer value, which causes a crash during printing. The second ranked predicate indicates that a certain count is negative. The count is subsequently passed to the function `memmove()` and causes a crash.

The fourth predicate is a secondary indicator of the last bug in EXIF. The primary indicator of the bug, `o + s > buf_size`, is ranked number 1 in predicate 4's predicate cluster. When this condition is true, the program neglects to allocate a chunk of memory, which crashes the program at a later stage. Out of 2211 failed runs in the EXIF dataset, only 12 crashed due to this bug. Hence its predictor is ranked behind the predictors for the other two bugs.

Overall, our multi-bug algorithm successfully clusters failed runs by their bugs, and often selects direct predictors of those bugs. Furthermore, given these run clusters, we can apply simple single-bug algorithms such as univariate hypothesis testing to rank and cluster correlated predicates. We find that these predicate clusters are often useful in interpreting the output of our debugging algorithm. It is worth noting that, without the run clusters, simple univariate algorithms would not have captured the correct bug predictors in our datasets.

4.3. Comparison to Other Algorithms

We compare our algorithm with two statistical debugging techniques presented in recent literature. In earlier work (Liblit et al., 2005), we proposed a predicate ranking algorithm based on the harmonic mean of predicate sensitivity and specificity. The algorithm then performs a heuristic "projection" step in which runs that have already been accounted for by the top predicate are eliminated. Liu et al. (2005) present the SOBER algorithm, which aims to solve the single-bug problem using a test derived from one-sample hypothesis testing.

We first compare the three algorithms on MOSS, RHYTHMBOX, and EXIF. The harmonic mean projection method selects predicates of comparable quality to the bi-clustering method in all three cases. The

[3] Let π_f denote the average truth probability of a predicate in the failed run cluster, and π_s that in the set of successful runs. We rank predicates according to the test statistic of the two-sample Bernoulli test for $\pi_f > \pi_s$.

SOBER single-bug algorithm fares much worse: for MOSS, it selects 25 sub-bug predictors for bug #1, followed by 4 predictors for bug #4, followed by more sub-bug predictors for bug #1.

To compare performance more systematically, we estimate the amount of programmer effort required to find bugs using each algorithm. Starting from the top ranked predictor, we model the programmer as performing a breadth-first search in the program dependence graph until reaching the buggy line(s) of code (Cleve & Zeller, 2005; Renieris & Reiss, 2003). The effort required is the percentage of code examined during this search. An implementation of this metric was provided by Holger Cleve and runs atop CodeSurfer, provided by GrammaTech, Inc.

We apply this comparison to the Siemens test suite (Hutchins et al., 1994), which contains 130 single-bug variants of 7 programs. This suite cannot gauge the effectiveness of multi-bug debugging algorithms, but it is currently the most widely used large benchmark for bug-hunting tools. We find that SOBER performs slightly better up to 7% of code examined, while above this point our bi-clustering algorithm offers superior performance. Overall, the bi-clustering algorithm requires less code examination on average than the other two algorithms on 130 tested programs, as is demonstrated in the following performance matrix:

	better	worse	same
Bi-cluster vs. SOBER	65	60	5
Bi-cluster vs. Project	70	53	7
SOBER vs. Project	66	53	11

5. Conclusions

In this paper, we present a systematic approach to statistical debugging of software programs in the presence of multiple bugs. Unlike its simpler single-bug sibling, the multi-bug problem is compounded by issues of sampling sparsity and complex inter-predicate relationships. Our algorithm specifically targets the pitfalls of simpler algorithms, and is proven to work well empirically on real world programs. Furthermore, the probability inference and collective voting framework could potentially be adjusted to accommodate more general bugs and predicate settings.

References

Cleve, H., & Zeller, A. (2005). Locating causes of program failures. *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005).* St. Louis, Missouri.

Dhillon, I. S. (2001). Co-clustering documents and words using bipartite spectral graph partitioning. *Proceedings of The Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD).* San Francisco, California.

Hartigan, J. A. (1972). Direct clustering of a data matrix. *Journal of the American Statistical Association, 67*, 123–129.

Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994). Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proc. 16th Int. Conf. Software Engeineering (ICSE'94)* (pp. 191–200).

Liblit, B., Aiken, A., Zheng, A. X., & Jordan, M. I. (2003). Bug isolation via remote program sampling. *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation.* San Diego, California.

Liblit, B., Naik, M., Zheng, A. X., Aiken, A., & Jordan, M. I. (2005). Scalable statistical bug isolation. *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation.* Chicago, Illinois.

Liu, C., Yan, X., Fei, L., Han, J., & Midkiff, S. P. (2005). SOBER: Statistical model-based bug localization. *Proceedings of the Fifth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE-05).*

Ng, A. Y., Jordan, M. I., & Weiss, Y. (2002). On spectral clustering: Analysis and an algorithm. *Advances in Neural Information Processing Systems 14.* Cambridge, MA: MIT Press.

Renieris, M., & Reiss, S. P. (2003). Fault localization with nearest neighbor queries. *Proc. 21st Int. Conf. on Automated Software Engineering (ASE'03)* (pp. 30–39). IEEE Computer Society.

Zheng, A. X., Jordan, M. I., Liblit, B., & Aiken, A. (2004). Statistical debugging of sampled programs. *Advances in Neural Information Processing Systems 16.* Cambridge, MA: MIT Press.