Statistical Software Debugging

by

Alice Xiaozhou Zheng

B.A. (University of California, Berkeley) 1999

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

Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Michael I. Jordan, Chair
Professor Peter Bartlett
Professor Bin Yu

Fall 2005

The dissertation of Alice Xiaozhou Zheng is approved:

Professor Michael I. Jordan, Chair	Date
Professor Peter Bartlett	Date
Professor Bin Yu	Date

University of California, Berkeley

Fall 2005

Statistical Software Debugging

by

Alice Xiaozhou Zheng

Abstract

Statistical Software Debugging

by

Alice Xiaozhou Zheng

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Science

University of California, Berkeley

Professor Michael I. Jordan, Chair

Statistical debugging is a combination of statistical machine learning and software debugging. Given sampled run-time profiles from both successful and failed runs, our task is to select a small set of program predicates that can succinctly capture the failure modes, thereby leading to the locations of the bugs. Given the diverse nature of software bugs and coding structure, this is not a trivial task.

We start by assuming that there is only one bug in the program. This allows us to concentrate on the problem of non-deterministic bugs. We design a utility function whose components may be adjusted based on the suspected level of determinism of the bug. The algorithm proves to work well on two real world programs.

The problems becomes much more complicated once we do away with the single-bug assumption. The original single-bug algorithm does not perform well in the presence of multiple bugs. Our initial attempts at clustering fall short of an effective solution. After identifying the main problems in the multi-bug case, we present an iterative predicate scoring algorithm. We demonstrate the algorithm at work on five real world programs, where it successfully clusters runs and identifies important predicates that clearly point to many of the underlying bugs.

Professor Michael I. Jordan, Chair Date

Acknowledgements

I would like to thank my advisor, Dr. Michael I. Jordan, for his support and guidance throughout the years. He has the knowledge for doing what needs to be done and the wisdom to let people do what they think needs to be done. He did not tell me which way to go, but instead paved the way and patiently waited for me to find it. Finding the way was not easy but I am glad that Mike gave me the chance to do so early on, while I still had his support as a safety net. His understanding and encouragement is what made this work possible.

I would like to thank my dear friend and collaborator Ben Liblit. Little did we know that a chance meeting four years ago during a Microsoft Research internship would blossom into a wonderful friendship and a long and fruitful collaboration. His insight and perspicacity still continues to astound me. Ben, you're a role model for geeks the world around! I hope we will collaborate again at some point in the future.

To Alex Aiken and Mayur Naik, our other two collaborators for the project: thank you for carrying the burden at times when no one else could, and for continuing the work when others – or rather, I – were discouraged and out of ideas.

I thank my family for making it possible for me to go this far in my pursuit of knowledge and understanding. They have always been there and have given me steadfast support, sometimes at the sacrifice of their personal freedoms. My gratitude towards them alone are enough to fill a separate thesis.

I would like to thank my friends: Alexandra, Ali, Andrew, Dave, Emory, Gene, Hanna, Michele, Mike, Misha, Vassilis, the entire family of the Engelhardts, and many others. They taught me about life. And special thanks to my "sister" Barbara, for printing out this thesis, for the conversations and support, and for all the cookies.

Contents

Chapter 1

Introduction

Machines have been inextricably linked with our lives since the beginning of history. From the simple flint hunting tools to the modern computer, machines have become smarter over time. They have become more automated, and their tasks increasingly abstract. In the field of machine learning, researchers concentrate on the goal of making self-reliant machines that are able to learn from data and perform detail analysis work that is often difficult for the human eye.

Machine learning algorithms utilize tools from statistics, optimization theory, and engineering. From a certain perspective, machine learning as a discipline is fundamentally application-oriented. On the one hand, it has the rigorous theoretical underpinnings from statistics and applied mathematics. On the other hand, it is useful in fields as diverse as bioinformatics, text, image, and music analysis, social network analysis, astrophysics, and many more. Machine learning researchers continue to find exciting application niches in emerging fields.

All machines eventually fail. For example, software program users are familiar with failures caused by software bugs. Programs often crash in the middle of an important task, or, more commonly, they may return incorrect output. Not only are

software bugs inconvenient for the user, they also adversely affect productivity and delay the completion of important tasks. Persistent software bugs and system failures may have profound social, economic, and security impacts on our society.

No one wants to release buggy software, but software companies often lack the time and resources to catch all of the major bugs prior to release. Thus we have the situation where, on the one hand, there are software users who need bug-free software, and on the other hand, there are programmers and companies who need their help. It would be advantageous for both sides to join their efforts in battling software bugs. Test engineers would be able to find more bugs with less time, and users would be able to direct debugging efforts towards important bugs effecting the most people.

There have been previous attempts at user-assisted bug-finding. Both Microsoft Windows and the Mozilla web browser have deployed crash report feedback mechanisms on their software. Each time the program crashes, a crash report is sent back to a server. The report contains basic information such as the stack trace and names of loaded library modules at the time of crash. Programs with a large user base can thus obtain thousands of crash reports everyday, many of which reveal the same problems. Companies have used these reports to guide their debugging efforts.

But we can do much more. With statistical debugging, we can take user-assisted debugging one step further. We collect reports of incorrect behavior, as well as those of correct behavior. Through statistical analysis and comparison of the two sets, we aim to automatically isolate the locations of bugs in the program.

In this thesis, I present some findings in the statistical debugging project. It has been a collaborative effort between software engineering (Ben Liblit, Alex Aiken, and Mayur Naik), and machine learning (Michael Jordan and myself). In the span of two and a half years, we have set up a small public deployment of our program report collection framework, and have wrestled with the complicated task of finding multiple bugs in real world programs.

The thesis is organized as follows. chapter 2 presents a brief survey of existing work in the realm of automatic software debugging. I introduce the run-time report collection framework in chapter 3, emphasizing on the nature of the data we obtain. In chapter 4, I present an algorithm that is useful for locating single bugs. We observe that the algorithm works well on programs containing only one bug, but does not work as well in the presence of multiple bugs. I describe important characteristics of the multi-bug problem in chapter 5, which lead to specific requirements of the multi-bug analysis algorithm. I present a multi-bug algorithm in chapter 6, and discuss empirical results on real-world programs in chapter 8. I conclude with a discussion of possible extensions and future work in chapter 9.

Chapter 2

Related Work

The work presented in this thesis falls under the category of dynamic program analysis. At the other end of the spectrum, there is currently a great deal of interest in applying static analysis to improve software quality [Gould *et al.*, 2004; Johnson and Wagner, 2004; Flanagan *et al.*, 2002; Henzinger *et al.*, 2002]. While we firmly believe in the use of static analysis to find and prevent bugs, our dynamic approach has advantages as well. A dynamic analysis can observe actual run-time values, which is often better than either making a very conservative static assumption about run-time values for the sake of soundness or allowing some very simple bugs to escape undetected. Another advantage of dynamic analysis, especially one that mines actual user executions for its data, is the ability to assign an accurate importance to each bug. Additionally, a dynamic analysis that does not require an explicit specification of the properties to check can find clues to a very wide range of errors, including classes of errors not considered in the design of the analysis.

The Daikon project [Ernst *et al.*, 2001] monitors instrumented applications to discover likely program invariants. It collects extensive trace information at run time and mines traces offline to accept or reject any of a wide variety of hypothesized

candidate predicates. The DIDUCE project [Hangal and Lam, 2002] tests a more restricted set of predicates within the client program, and attempts to relate state changes in candidate predicates to manifestation of bugs. Both projects assume complete monitoring, such as within a controlled test environment. Our goal is to use lightweight partial monitoring, suitable for either testing or deployment to end users.

Software tomography as realized through the GAMMA system [Bowring *et al.*, 2002; Orso *et al.*, 2003] shares our goal of low-overhead distributed monitoring of deployed code. GAMMA collects code coverage data to support a variety of code evolution tasks. Our instrumentation exposes a broader family of data- and control-dependent predicates on program behavior and uses randomized sparse sampling to control overhead. Our predicates do, however, also give coverage information: the sum of all predicate counters at a site reveals the relative coverage of that site.

Delta Debugging [Zeller and Hildebrandt, 2002] compares inputs of instances of successful runs with those of failing runs. It repeatedly tests the program with different inputs until it narrows down the input string to a minimum set that causes the program to fail. Zeller has extended the same technique to internal program states, thereby identifying cause-effect chains of failure [Zeller, 2002]. Recently, Holger and Zeller have begun to further analyze cause-effect chains in order to better isolate causes of program failures [Zeller, 2002].

Efforts to directly apply statistical modeling principles to debugging have met with mixed results. Early work in this area by Burnell and Horvitz [Burnell and Horvitz, 1995] uses program slicing in conjunction with Bayesian belief networks to filter and rank the possible causes for a given bug. Empirical evaluation shows that the slicing component alone finds 65% of bug causes, while the probabilistic model correctly identifies another 10%. This additional payoff may seem small in light of the effort, measured in man-years, required to distill experts' often tacit knowledge into a formal belief network. However, the approach does illustrate one strategy for

integrating information about program structure into the statistical modeling process.

In more recent work, Podgurski *et al.* [2003] apply statistical feature selection, clustering, and multivariate visualization techniques to the task of classifying software failure reports. The intent is to bucket each report into an equivalence group believed to share the same underlying cause. Features are derived offline from fine-grained execution traces without sampling; this approach reduces the noise level of the data but greatly restricts the instrumentation schemes that are practical to deploy outside of a controlled testing environment. As in our own earlier work, Podgurski uses logistic regression to select features that are highly predictive of failure. Clustering tends to identify small, tight groups of runs that do share a single cause but that are not always maximal. That is, one cause may be split across several clusters. This problem is similar to covering a bug profile with sub-bug predictors (see section 5.2).

In contrast, current industrial practice uses stack traces to cluster failure reports into equivalence classes. Two crash reports showing the same stack trace, or perhaps only the same top-of-stack function, are presumed to be two reports of the same failure. This heuristic works to the extent that a single cause corresponds to a single point of failure, but our experience with several real world programs suggests that this assumption may not often hold. In the MOSS program, we find that only bugs #2 and #5 have truly unique "signature" stacks: a crash location that is present if and only if the corresponding bug was actually triggered. These bugs are also our most deterministic. Bugs #4 and #6 also have nearly unique stack signatures. The remaining bugs are much less consistent: each stack signature is observed after a variety of different bugs, and each triggered bug causes failure in a variety of different stack states. In the programs RHYTHMBOX and EXIF, bugs caused crashes so long after the bad behavior that stacks were of limited use or no use at all.

Studies that attempt real-world deployment of monitored software must address a host of practical engineering concerns, from distribution to installation to user support

to data collection and warehousing. Elbaum and Hardojo [2003] have reported on a limited deployment of instrumented Pine binaries. Their experiences have helped to guide our own design of a wide public deployment of applications with sampled instrumentation, presently underway [Liblit *et al.*, 2004].

For some highly available systems, even a single failure must be avoided. Once the behaviors that predict imminent failure are known, automatic corrective measures may be able to prevent the failure from occurring at all. The Software Dependability Framework (SDF) [Gross *et al.*, 2003] uses multivariate state estimation techniques to model and thereby predict impending system failures. Instrumentation is assumed to be complete and is typically domain-specific. Our algorithm could also be used to identify *early warning* predicates that predict impending failure in actual use.

Our bug-finding algorithm has evolved over the course of this project. The single-bug algorithm has been previously published at PLDI and NIPS. Earlier versions of the multi-bug algorithm are very different from that presented in this thesis and have been published at PLDI. These papers have generated some interest in this area of research. Notably, Liu *et al.* [2005] apply a hypothesis test on the truth frequency of predicates in failed vs. successful runs. Their algorithm yielded better results on a different set of programs with complete observations (no down-sampled predicate counts). Up to now we have assumed that binary true/false predicate counts are more suited for bug-finding than the actual number of truth counts. Their experiments, however, may serve as evidence to the contrary.

Chapter 3

Data Generation and Collection

We begin the description of our automatic debugging system with the predicate sampling framework. The sampling process generates truth counts of program predicates, which are analyzed in subsequent parts of the system.

In existing crash feedback systems, crash reports include information such as the program stack at the time of crash and the set of loaded libraries. Such coarse-grained information may be easy to record and store, but they often do not contain enough details to be useful. Finer-grained debugging information would be able to indicate the problems more precisely. Thus the ultimate goal of our predicate sampling framework is to collect detailed information that presents an unbiased view of the run-time internal states of the program.

We would like to gather as much information as possible from the program while it is running. But the data collection must not incur too much cost to the user, because it is ultimately up to the user to adopt the system and provide us with program reports. Few users would tolerate a run-time information-collection system that greatly effects the speed of the program. To save time we must sample the predicates. This means the collected data from a single run will be sparse. To counter the effects of sampling,

we will need more runs of the program, as well as good analysis algorithms that take sampling into effect.

3.1 Instrumentation Schemes

The program report collection process begins with a source-to-source transformation of the program. Given the original source code, a long list of "questions" is automatically generated and inserted into appropriate places in the source. These questions query the state of program variables. The *instrumented* program then executes these queries and collects the answers.

There are many kinds of questions one might ask the program. Often times, the flow of the program contains important debugging information. For instance, at branch statements, it might be useful to know which branch was ultimately taken. However, recording the details of each branch decision would require a lot of storage space. Instead it might be sufficient to record only the number of times a certain branch conditional returns the answer `TRUE` vs. `FALSE`. We can insert a line of code for each branch statement and record this information.

Return values of functions can also be important problem indicators. In particular, C functions often use return values to indicate success or failure. For example, the `fopen()` function either returns a valid file pointer, or the value zero if an error occurs. Such error indicators are often neglected by careless programmers. Thus, a record of the return values of each and every function call may yield important clues about what went wrong during a run of the program. Again, due to space considerations, we settle for a record of the total number of times each function returns a value greater than, equal to, and less than zero.

Array access out-of-bounds errors are common programming mistakes with far-reaching consequences. The Java programming language automatically checks every

array access, but C programs have no protection against attempts to read or write beyond the correct boundaries of an array. When a piece of memory is overwritten by mistake, the consequences may not be apparent until much later. This delay in time makes such mistakes very difficult to track down. To catch such mistakes, a programmer might add checks for all array indices to ensure that they are within their legal range. But this is labor-intensive. Our statistical debugging framework, on the other hand, can automatically insert questions at appropriate places, but would have a difficult time identifying array indices and the correct array lengths. Thus instead, we opt to compare every pair of integer valued program variables, and record the number of times one is greater than, equal to, or less than another. This gives us very detailed information that may also be useful for catching other types of bugs.

To recapitulate, we introduce three *instrumentation schemes*, or general suites of queries.

branches: at each branch statement, record the number of times the program takes the true branch and the false branch;

return values: for each function call returning a character, integer, or pointer, record the number of times the return value is greater than, less than, or equal to zero;

scalar-pairs: at each integer-valued assignment of the form `x = ...`, compare `x` to each integer variable and program constant in scope, and record the number of times it is greater than, equal to, or less than `x`.

While other instrumentation schemes have been considered, the three listed above form the backbone of the bulk of our experiments. Once the instrumentation scheme is chosen, all queries are generated automatically. The branch scheme checks every

branch statement, the return value scheme every function call, and so on. No manual effort is involved. The test engineer need only to decide which instrumentation schemes to include, and the rest is done automatically.

We define each line of inserted instrumentation code to be an *instrumentation site*. Each instrumentation site contains a *query* to the program, such as "did the last branch conditional statement evaluate to true or false." The answer to the query is used to update one of several *predicate counters*. For instance, each branch instrumentation site is associated with two predicates: `TRUE` and `FALSE`. If the query returned `TRUE`, then the `TRUE` predicate counter is incremented by one, otherwise the `FALSE` predicate counter is incremented.

3.2 The Predicate Sampling Framework

Depending on the size and complexity of the original program, the instrumented version might contain thousands to millions of instrumentation sites. The program might take several times longer to run if it were to execute every query it encounters. One way to save time is to execute only a subset of the queries in each run. We devise the following random instrumentation site sampling process, which guarantees an unbiased view of the internal states of the program.

Imagine that the program tosses a random coin every time it comes across an instrumentation site. If the coin comes up heads, then the program executes the query and records the answer; otherwise the site is skipped. Thus, as the program runs, a Bernoulli trial of independent coin tosses decides whether or not to take each sample. Suppose that the coin has a Ber(d) distribution, and suppose that a certain site is reached exactly once in every run of the program. Then on average, we would expect to see one sample of the site per $1/d$ runs.

Generating a random bit every time we reach an instrumentation site still requires

a lot of computation resources. Recall that, in a Bernoulli trial, the arrival time of the next head is geometrically distributed. Hence instead of generating individual random bits, we can generate a Geo(d) distributed counter. Each site decrements the counter by one. When the counter reaches zero, a sample is taken and a new counter is generated.

Furthermore, there are ways of decrement the counter in bulk when necessary. For example, we determine, at compile time, that there are n sites contained in a basic block (i.e., a block of code with no branches or function calls to the outside). Upon reaching the block at run time, if the current value of the counter is greater than n, then the program may simply decrement the counter by n and execute a fast version of the code that does not contain instrumentation. Only when the counter value is smaller than or equal to n would the program need to step through each site individually. (Interested readers may refer to Liblit [2004] for details on implementation.)

The sampling rate d controls the trade-off between run time overhead and data sparsity. Smaller values of d incur less overhead, but result in sparser data. In Liblit [2004], Liblit experiments with various sampling rates on a set of twelve benchmark programs. Some of those results are included here. Table 3.1 summarizes the amount of additional CPU time taken by the instrumented program compared with the uninstrumented version.

This experiment demonstrates an interesting phenomenon. When the sampling rate is not sparse enough, it is better to just always take the sample, i.e., setting the sampling rate to 1. This is because when a sample is always taken, the program does not need to generate the geometric counters or check to see if it has reached zero. Such checks involve costly branch statements. In some cases, decreasing the sampling rate down to $1/1000$ keeps the overhead at a tolerable level.

Table 3.1: Instrumentation overhead in benchmark programs [Liblit, 2004].

	Overhead for Sampling Rate				
Benchmark	$1/1$	$1/10$	$1/100$	$1/1000$	$1/10000$
bh	576%	5205%	731%	126%	28%
bisort	570%	2309%	301%	69%	44%
compress	2049%	9075%	1202%	191%	74%
em3d	226%	1118%	136%	35%	25%
health	41%	230%	22%	5%	1%
ijpeg	1422%	6627%	869%	149%	46%
mst	86%	452%	62%	15%	10%
perimeter	209%	1565%	246%	99%	76%
power	23%	406%	55%	19%	16%
treeadd	69%	524%	62%	12%	7%
tsp	140%	359%	51%	9%	4%
vortex	804%	4708%	679%	130%	60%

3.3 Non-Uniform Sampling

Sampling reduces run time overhead, but leads to sparse data. Such effects are especially apparent at rarely reached instrumentation sites. However, in many situations, such rarely reached sites may provide particularly important clues about the bug. Thus the quality of the recorded data may be greatly improved through *non-uniform* sampling of the sites. With non-uniform sampling, rare sites may be given a much higher sampling rate than sites that are often reached. Since they are seldom reached, an increase in sampling rate would likely have little consequence for the run-time overhead.

There are a few different implementation choices for non-uniform sampling. For example, one might employ multiple countdown counters, each with a different geometric distribution. Alternatively, each site may choose to decrement the countdown by a number other than 1. Sites with higher weight would have a higher sampling probability. Either way, non-uniform sampling requires a sampling plan for each in-

strumentation site. Such a plan may be gathered from a few training runs where the sampling probability set to 1. The sampling probability of each site can then be set to the inverse of its observed frequency.

The run time overhead of non-uniform sampling depends on the particular sampling plan of the sites. We have not systematically investigated the effects of non-uniform sampling on run time overhead. But due to the potential benefits for the analysis algorithm, all the datasets used in our experiments are generated with non-uniform sampling.

3.4 The Cooperative Bug Isolation Project

There is an on-going public deployment of our program sampling framework. The Cooperative Bug Isolation project (CBI) (http://www.cs.wisc.edu/cbi/) directly distributes instrumented binaries of open source programs to end-users [Liblit *et al.*, 2004]. This is an opt-in system where users have the option of choosing to send a compressed run-time profile back to our server at the end of every program run.

Currently, six open-source programs are instrumented and ready for download. Evolution is an email-calender program; Gaim is an instant messaging tool; The GIMP is an image manipulation program much like Adobe Photoshop; Nautilus is a session-management and file-navigation tool; RHYTHMBOX is an MP3 music player; SPIM is an assembly language interpreter.

To date, we have collected a number of run-time profiles for these programs, but the dataset is not yet large enough for comprehensive testing of our algorithms. The bulk of our experiments are conducted on generated data from five programs.

Table 3.2: Summary statistics for debugging datasets.

		Runs			
	Lines of Code	Successful	Failing	Sites	Predicate Counts
CCRYPT	5276	20,684	10,316	9948	58,720
BC	14,288	23,198	7802	50,171	298,482
MOSS	6001	26,239	5505	35,223	202,998
RHYTHMBOX	56,484	12,530	19,431	14,5176	857,384
EXIF	10,588	30,789	2211	27,380	156,476

3.5 The Datasets

For our experiments, we generate data from five real-world programs: BC, CCRYPT, MOSS, RHYTHMBOX, and EXIF. All three instrumentation schemes are included. For each run, a sequence of valid command-line arguments are chosen randomly to simulate actual user choices. The instrumented program binary collects data with the sampling probability set to 1 for all sites. The predicates counts are down sampled afterwards according to non-uniform sampling rates gathered based on 1000 training runs. All of our experiments are performed on sampled data.

Table 3.2 contains some relevant statistics of the dataset. We will introduce each of the programs in turn.

3.5.1 CCRYPT

CCRYPT is a command-line tool for file encryption and decryption. CCRYPT 1.2 has a known input validation bug. At one point in the code the program attempts to write a file. If the file already exists, the user is presented with a prompt asking for confirmation that the file should be overwritten; if the user input is EOF (i.e., the user hits `Enter` without typing anything) the program will crash. Compared to the non-deterministic bug in BC, this bug is deterministic and much easier to find for a

human. Thus it should also be an easy target for the statistical debugging algorithm.

3.5.2 BC

BC is a Unix command-line calculator program. GNU BC 1.06 has a previously reported buffer overrun bug. In the memory reallocation routine `more_arrays()`, one of the memory-writing loops uses the wrong array length. When this bug is triggered, the program overwrites memory area outside of its legal limit, which crashes the program in a non-deterministic fashion. Sometimes the program hits the overwritten area and crashes. But in about four out of ten runs, the program gets lucky and never re-uses the trampled memory again, and thus manages to complete successfully. We call this a *non-deterministic* bug because it does not trigger a certain outcome.

3.5.3 MOSS

MOSS is a widely used service for detecting plagiarism in software [Schleimer *et al.*, 2003]. As a validation experiment for our multi-bug algorithm, we add nine bugs to MOSS. Six of these were previously discovered and repaired bugs in MOSS that we reintroduced. The other three were variations on three of the original bugs, to see if our algorithms could discriminate between pairs of bugs with very similar behavior but distinct causes. We briefly describe each of these nine bugs.

1. We reintroduced a bug that causes the number of lines in C-style multi-line comments to be counted incorrectly. This bug causes incorrect output in certain circumstances: an option to match comments must be on (normally MOSS ignores comments) and there must be matching multi-line comments that affect the output.

2. We removed a check for a null `FILE` pointer. This is not originally a MOSS

bug; it is exactly analogous to the CCRYPT bug (subsection 3.5.1).

3. We removed an array bounds update in the routine for loading preprocessed data from disk. The program behaves normally unless the function is called a second time, in which case previously loaded data may be partially overwritten. This bug has unpredictable effects and was particularly difficult to find originally.

4. We removed a size check that prevents users from supplying command-line arguments that could cause the program to overrun the bounds of an array.

5. For historical reasons, MOSS handles Lisp programs differently from all other languages. We removed a end-of-list check in the Lisp-specific code.

6. For efficiency MOSS preallocates a large area of memory for its primary data structure. When this area of memory is filled, the program should fail gracefully. We removed an out-of-memory check.

7. MOSS has a routine that scans an array for multiple copies of a data value. We removed the limit check that prevents the code from searching past the end of the array. This bug did not occur in MOSS; it is intended to be a more frequently occurring version of bug 8.

8. A buffer overrun, this bug was never known to have caused a failure in MOSS. It was discovered originally by a code review.

9. This bug is a variant of bug 4, but involves a different command-line argument and a different array.

In summary, there are eight various crashing bugs: four buffer overruns, a null file pointer dereference in certain cases, a missing end-of-list check in the traversal of a hash table bucket, a missing out-of-memory check, and a violation of a subtle

invariant that must be maintained between two parts of a complex data structure. In addition, some of these bugs are non-deterministic and may not even crash when they should.

The first bug—incorrect comment handling in some cases—only causes incorrect output, not a crash. We include this bug in our experiment in order to show that bugs other than crashing bugs can also be isolated using our techniques, provided there is some way, whether by automatic self-checking or human inspection, to recognize failing runs. In particular, for our experiment we also ran a correct version of MOSS and compared the output of the two versions. This oracle provides a labeling of runs as "success" or "failure," and the resulting labels are treated identically by our algorithm as those based on program crashes.

3.5.4 RHYTHMBOX

RHYTHMBOX is an interactive, graphical, open source music player. Its code structure is complex, multi-threaded, and event-driven. Event-driven systems use event queues; each event performs some computation and possibly adds more events to some queues. This is an application where our approach has a definitely advantage over other static program analysis techniques. Most of the interesting things are happening in the event queues, which can only be captured using a run-time framework like our own.

RHYTHMBOX 0.6.5 contains at least two previously unknown bugs. The first bug is a race condition, where a previously freed object is called on by another event. This is probably due to mis-synchronization between different threads of the program. In some cases, the RHYTHMBOX shell player object continues receiving signals even after it has been destroyed. The bug manifests itself when the function call `monkey_media_player_get_uri()` returns `NULL`.

The second bug we uncovered has to do with dangling pointers. At some point, an timer object used in animating a disclosure widget is destroyed, but a pointer to the timer object is never updated. The program later tries to destroy the event pointed to by the old pointer, and subsequently crashes if the event ID has already been reclaimed by another object. This bug in fact uncovers a whole set of incorrect coding practices that occur over and over again in RHYTHMBOX. Often times, the event is destroyed, but the pointer remains, potentially wreaking havoc later on.

These bugs were previously unknown. We discovered both of them using an earlier version of our statistical debugging algorithm [Liblit *et al.*, 2005]. We have reported the bugs to the RHYTHMBOX developers and the reports have been enthusiastically received.

3.5.5 EXIF

EXIF is an open-source image processing program. We discovered three bugs in EXIF 0.6.9 using a statistical debugging algorithm [Liblit *et al.*, 2005]. First, the program crashes in the "-m" (machine readable) mode when a `NULL` value is passed to the `fprintf()` function. Secondly, removing the thumbnail from a non-JPEG image leads to a negative byte count being passed to `memmove()`, which in turn causes a crash. Thirdly, manipulating thumbnails in Canon images causes the program to crash. In the last case, the branching condition `o + s > buf_size` being true causes the function `exif_mnote_data_canon_load()` to return without allocating space for certain data structures. The program crashes later on when those data structures are to be used.

Chapter 4

Catching a Single Bug

4.1 Possible Simplifications

Software bugs can occur in many forms and under diverse circumstances. A few simplifying assumptions are necessary before we can start designing statistical debugging algorithms.

Simplification is possible along several directions. For example, we may be tempted to assume that sampling has no qualitative effect on the data, and hope that an algorithm that works well on full data may work equally well on a larger set of sampled data. But this is not the case. Through our experiments, we learned that sampling causes numerous unanticipated problems for the analysis algorithm. A statistical debugging algorithm needs to compare predicates against each other and pick out the most "informative" ones. But because each instrumentation site is reached a different number of times during the run, sampling affects each predicate differently. Thus we must take care when comparing predicates against each other. In our experience, it is often very easy to design algorithms that work for unsampled data, but such algorithms become a lot less effective when given sampled data.

One may also make simplifying assumptions about the type of bugs to catch. This is implicitly determined by the instrumentation schemes we choose. The scalar-pairs scheme, for example, is designed to catch array-overrun type bugs. We cannot catch bugs that cannot be detected by the instrumentation.

Lastly, we can make assumptions about the number of bugs in the program. We start with a single-bug assumption: all the failed runs are caused by the same bug. This is by no means a realistic assumption. But such homogeneity makes the problem much easier to solve. By starting out on the single-bug path, we hope to be able to extrapolate about the more general multi-bug case. Whether this is possible remains to be seen. For now, we work on catching a single bug.

4.2 The Approach

We choose to approach the problem under the classification and feature selection setting. This is a classic setup in machine learning. But we find that a few changes are necessary in order to navigate the twists and turns of software debugging. We choose to work under a decision-theoretic framework because it is flexible enough to adapt to the particular requirements of the problem at hand.

We can easily set up a classification task based on the crash and success labels. Our primary goal, however, is that of feature selection [Blum and Langley, 1997]. It has been noted that the goals of feature selection do not always coincide with that of classification [Guyon and Elisseeff, 2003]. In our case, classification is but the means to an end. Good classification performance assures the user that the system is working correctly, but one still has to examine the selected features to see that they make sense. In the debugging problem, we only care about features that correctly predict failures. Hence, instead of working in the usual maximum likelihood setting for classification and regularization, we define and maximize a more appropriate utility

function.

4.3 Characteristics of the Single-Bug Problem

We concentrate on isolating the bugs that are caused by the occurrence of a small set of features, i.e., predicates that are always true when a crash occurs.[1] Thus we want to identify the predicate counts that are positively correlated with the program crashing. In contrast, we do not care much about the features that are highly correlated with successes. This makes our feature selection an inherently one-sided process.

Due to sampling effects, it is quite possible that a feature responsible for the ultimate crash may not have been observed in a given run. This is especially true in the case of "quick and painless" deaths, where a program crashes very soon after the actual bug occurs. Normally this would be an easy bug to find, because one wouldn't have to look very far beyond the crashing point at the top of the stack. However, this is a challenge for our approach, because there may be only a single opportunity to sample the buggy feature before the program dies. Thus many crashes may have an input feature profile that is very similar to that of a successful run. From the classification perspective, this means that false negatives are quite likely.

At the other end of the spectrum, if we are dealing with a *deterministic bug*[2], false positives should have a probability of zero: if the buggy feature is observed to be true, then the program has to crash; if the program did not crash, then the bug must not have occurred. Therefore, for a deterministic bug, any false positives during the training process should incur a much larger penalty compared to any false negatives.

[1] There are bugs that are caused by non-occurrence of certain events, such as forgotten initializations. We choose not to deal with this kind of bugs here.

[2] A bug is *deterministic* if it crashes the program every time it is observed. For example, dereferencing a null pointer would crash the program without exception.

4.4 Designing the Utility Function

We need to design a utility function that can handle all of the above stated characteristics. Let crashes be labeled with a 1, and successes 0. Let (x, y) denote a data point, where x is an input vector of non-negative integer counts, and $y \in \{0, 1\}$ is the output label. Let $f(x; \theta)$ denote a classifier with parameter vector θ. There are four possible decision outcomes: the true positive ($y = 1$ and $f(x; \theta) = 1$), the true negative ($y = 0$ and $f(x; \theta) = 0$), the false negative ($y = 1$ and $f(x; \theta) = 0$), and the false positive ($y = 0$ and $f(x; \theta) = 1$). In the general form of utility maximization for classification (see, e.g., [Lehmann, 1986]), we can define separate utility functions for each of the four cases.

We first select the functional form of the classifier. The actual distribution of input features X is determined by the software under examination, hence it is difficult to specify and highly non-Gaussian. We choose $f(x; \theta)$ to be a discriminative classifier. Assuming that the more abnormalities there are, the more likely it is for the program to crash, it is reasonable to use a classifier based on a linear combination of features. Let $z = \theta^T x$, where the x vector is now augmented by a trailing 1 to represent the intercept term. We use the logistic function $\mu(z)$ to model the class conditional probability:

$$P(Y = 1 \mid z) := \mu(z) = 1/(1 + e^{-z}).$$

The decision boundary is set to $1/2$, so that $f(x; \theta) = 1$ if $\mu(z) > 1/2$, and $f(x; \theta) = 0$ if $\mu(z) \leq 1/2$.

Let u_1, u_2, u_3, and u_4 denote the utility functions for each of the four decision scenarios. We choose the functional form to be logistic for all of the utilities, but add

an extra linear penalty term to u_4 for the case of false positives:

$$u_1(x;\theta) := u_2(x;\theta) := \delta_1 \log \mu(x;\theta) \tag{4.1}$$

$$u_3(x;\theta) := \delta_2 \log(1-\mu(x;\theta)) \tag{4.2}$$

$$u_4(x;\theta) := \delta_2 \log(1-\mu(x;\theta)) - \delta_3 \theta^T x \ . \tag{4.3}$$

The constants δ_1, δ_2, and δ_3 adjust the relative importance of the performance of the classifier in each decision scenario. We will discuss their interpretation in section 4.5.

The overall utility function is:

$$\begin{aligned} U(X,Y;\theta) \ = \ & u_1(X;\theta) Y \mathbb{I}_{\{f(X;\theta)=1\}} + u_2(X;\theta) Y \mathbb{I}_{\{f(X;\theta)=0\}} \\ & + u_3(X;\theta)(1-Y)\mathbb{I}_{\{f(X;\theta)=0\}} + u_4(X;\theta)(1-Y)\mathbb{I}_{\{f(X;\theta)=1\}} + v(\theta), \end{aligned}$$

where $\mathbb{I}_W$ denotes the indicator function for event W, and $v(\theta)$ is a regularization term, which may be interpreted in terms of a prior over the classifier parameters θ. In our experiments, we choose the regularization term to be the ℓ_1-norm of θ, which has the sparsifying effect of driving θ towards zero:

$$v(\theta) := -\lambda |\theta|_1^1 = -\lambda \sum_i |\theta_i|.$$

We maximize the empirical estimate of the expected utility:

$$\mathbb{E}_{P(X,Y)} \ U = \delta_1 y \log \mu + \delta_2 (1-y) \log(1-\mu) - \delta_3 \theta^T x (1-y) \mathbb{I}_{\{\mu > 1/2\}} - \lambda \|\theta\|_1^1 \ . \tag{4.4}$$

When $\delta_1 = \delta_2 = 1$ and $\delta_3 = 0$, Equation (4.4) is akin to the Lasso [Hastie *et al.*, 2001] (standard logistic regression with ML parameter estimation and ℓ_1-norm regularization). In general, however, our expected utility function weighs each class separately

via δ_i, and includes an additional penalty term for false positives.

Parameter learning is done using stochastic (sub)gradient ascent on the objective function. Besides being space efficient, such on-line methods also improve user privacy. Once the sufficient statistics are collected, the trial run can be discarded, thus obviating the need to permanently store any user's private data on a central server.

Equation (4.4) is concave in θ, but the ℓ_1-norm and the indicator function are non-differentiable at $\theta_i = 0$ and $\theta^T x = 0$, respectively. This can be handled by subgradient ascent methods[3]. In practice, we jitter the solution away from the point of non-differentiability by taking a very small step along any subgradient. This means that the values of θ_i will never be exactly zero during optimization. In practice, weights close enough to zero are essentially taken as zero. Only the few features with the most positive weights are selected at the end.

4.5 Interpretation of the Utility Function

Let us take a look at the utility functions defined in Equation (4.1), Equation (4.2), and Equation (4.3). They are essentially weighted logistic utilities with an added linear penalization for false positives. For the case of $Y = 1$, Figure 4.1(a) plots the function $\log_2 \mu(z) + 1$, a shifted and scaled version of u_1 and u_2. It is positive when z is positive, and approaches 1 as z approaches $+\infty$. It is a crude but smooth approximation of the indicator function for a true positive, $y\mathbb{I}_{\{\mu>1/2\}}$. On the other hand, when z is negative, the utility function is negative, acting as a penalty for false negatives. Similarly, Figure 4.1(b) plots the shifted and scaled utility functions for $Y = 0$. In both cases, the utility function has an upper bound of 1, so that the

[3]Subgradients are a generalization of gradients that are also defined at non-differentiable points. A subgradient for a convex function is any sublinear function pivoted at that point, and minorizing the entire convex function. For convex (concave) optimization, any subgradient is a feasible descent (ascent) direction. For more details, see, e.g., [Hiriart-Urruty and Lemarechal, 1993].

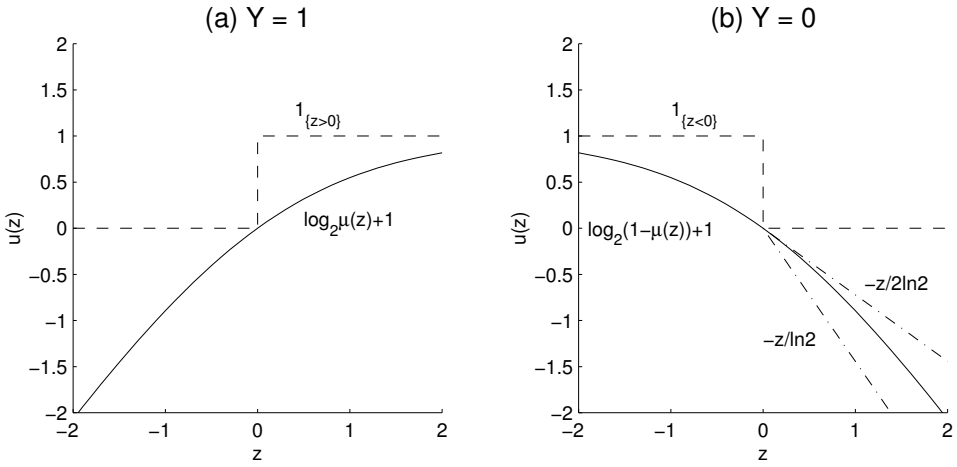

Figure 4.1: (a) Plot of the true positive indicator function and the utility function $\log_2 \mu(z) + 1$. (b) Plot of the true negative indicator function, utility function $\log_2(1 - \mu(z)) + 1$, and its asymptotic slopes $-z/\log 2$ and $-z/2\log 2$.

effect of correct classifications is limited. On the other hand, incorrect classifications are undesirable, thus their penalty is an unbounded (but slowly deceasing) negative number.

Taking the derivative $\frac{d}{dz}\log_2(1 - \mu(z) + 1) = -\mu(z)/\log 2$, we see that, when z is positive, $-1 \leq -\mu(z) \leq -1/2$, so $\log_2(1 - \mu(z)) + 1$ is sandwiched between two linear functions $-z/\log 2$ and $-z/2\log 2$. It starts off being closer to $-z/2\log 2$, but approaches $-z/\log 2$ asymptotically (see Figure 4.1(b)). Hence, when the false positive is close to the decision boundary, the additional penalty of $\theta^T x = z$ in Equation (4.3) is larger than the default false positive penalty, though the two are asymptotically equivalent.

Let us turn to the roles of the multiplicative weights. δ_1 and δ_2 weigh the relative importance of the two classes, and can be used to deal with imbalanced training sets where one class is disproportionately larger than the other [Japkowicz and Stephen, 2002]. Most of the time a program exits successfully without crashing, so we have to deal with having many more successful runs than crashed runs (see section 4.7).

Furthermore, since we really only care about predicting class 1, increasing δ_1 beyond an equal balance of the two data sets could be beneficial for feature selection performance. Finally, δ_3 is the knob of determinism: if the bug is deterministic, then setting δ_3 to a large value will severely penalize false positives; if the bug is not deterministic, then a small value for δ_3 allows for the necessary slack to accommodate runs which should have failed but did not. As we shall see in section 4.7, if the bug is truly deterministic, then the quality of the selected features will be higher for larger δ_3 values.

Our proposed solution to the single-bug problem is inspired by one of our earlier experiments [Liblit *et al.*, 2003]. We started with a few simple feature elimination heuristics and found them to be useful in locating bugs:

⟨**Elimination by universal falsehood**⟩**:** Discard any counter that is always zero, because it most likely represents an assertion that can never be true. This is a very common data preprocessing step.

⟨**Elimination by lack of failing example**⟩**:** Discard any counter that is zero on all crashes, because what never happens cannot have caused the crash.

⟨**Elimination by successful counterexample**⟩**:** Discard any counter that is non-zero on any successful run, because these are assertions that can be true without a subsequent program failure.

In the setting of our learning algorithm, ⟨elimination by universal falsehood⟩ is naturally incorporated as a preprocessing step. The other two heuristics are represented as well. If a feature x_i is never positive for any crashes, then its associated weight θ_i will only decrease in the maximization process. Thus it will not be selected as a crash-predictive feature. This handles ⟨elimination by lack of failing example⟩. Lastly, if a heavily weighted feature x_i is positive on a successful run in the training set, then the

classifier is more likely to result in a false positive. The false positive penalty term will then decrease the weight θ_i, so that such a feature is unlikely to be chosen at the end. Thus utility maximization also handles ⟨elimination by successful counterexample⟩.

4.6 Two Case Studies

We test our algorithm on CCRYPT and BC. CCRYPT serves as an example of a deterministic bug, and BC a non-deterministic bug. For simpler bugs, we need fewer runs to debug the program. We conduct our experiments on smaller datasets than those introduced in section 3.5. The quality of the datasets are the same, but here we pick the instrumentation scheme according to the suspected bug.

CCRYPT's sensitivity to `EOF` inputs suggests that the problem has something to do with its interactions with standard file operations. Thus the function return value instrumentation scheme may be the most helpful here. There are 570 call sites of interest which makes $570 \times 3 = 1710$ counters. We use the reports from 7204 trial runs, taken at a sampling rate of $1/100$. 1162 of the runs result in a crash. 6516 ($\approx 90\%$) of these trial runs are randomly selected for training, and the remaining 688 held aside for cross-validation. Out of the 1710 counter features, 1542 are constant across all runs, leaving 168 counters to be considered in the training process.

In the case of BC, we are interested in the behavior of all pointers and buffers. All pointers and array indices are scalars, hence we compare all pairs of scalar values. There are 30150 such scalar-pairs predicates, of which 2908 are not constant across all runs. Our small BC data set consists of 3051 runs with distinct random inputs, sampled at a uniform rate of $1/1000$. 2729 of these runs are randomly chosen as training set, 322 for cross-validation.

4.7 Results

We maximize the utility function in Equation (4.4) using stochastic subgradient ascent with a learning rate of 10^{-5}. In order to make the magnitude of the weights θ_i comparable to each other, the feature values are shifted and scaled to lie between $[0, 1]$, then normalized to have unit variance. There are four learning parameters, δ_1, δ_2, δ_3, and λ. Since only their relative scale is important, the regularization parameter λ can be set to some fixed value (we use 0.1). For each setting of δ_i, the model is set to run for 60 iterations through the training set, though the process usually converges much sooner. For BC, this takes roughly 110 seconds in MATLAB on a 1.8 GHz Pentium 4 CPU with 1 GB of RAM. The smaller CCRYPT dataset requires just under 8 seconds.

The values of δ_1, δ_2, and δ_3 can all be set through cross-validation. However, this may take a long time, plus we would like to leave the ultimate control of the values to the users of this tool. The more important knobs are δ_1 and δ_3: the former controls the relative importance of classification performance on crashed runs, the latter adjusts the believed level of determinism of the bug. We find that the following guidelines for setting δ_1 and δ_3 work well in practice.

1. In order to counter the effects of imbalanced datasets, the ratio of δ_1/δ_2 should be at least around the range of the ratio of successful to crashed runs. This is especially important for the CCRYPT data set, which contains roughly 32 successful runs for every crash.

2. δ_3 should not be higher than δ_1, because it is ultimately more important to correctly classify crashes than to forbid false positives.

As a performance metric, we look at the hold-out set confusion matrix and define the score as the sum of the percentages of correctly classified data points for each

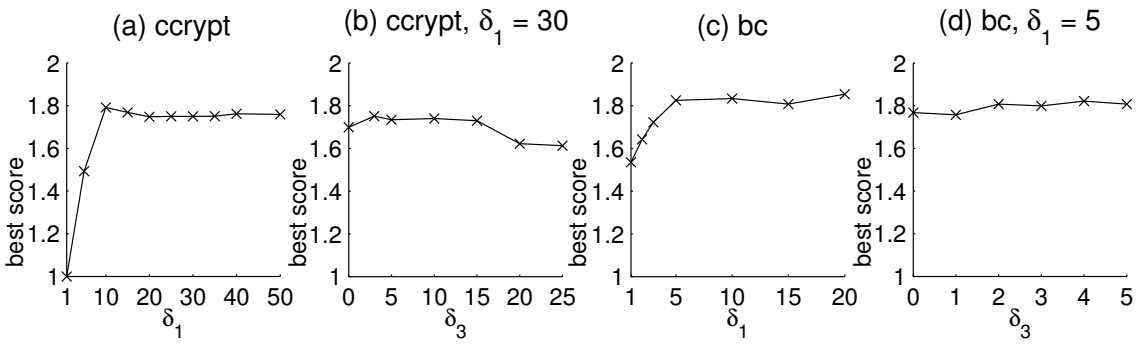

Figure 4.2: (a,b) Cross-validation scores for the CCRYPT data set; (c,d) Cross-validation scores for the BC data set. All scores shown are the maximum over free parameters.

class. Figure 4.2(a) shows a plot of cross-validation score for the CCRYPT data set at various δ_1 values, maximized over δ_2 and δ_3. It is clear from the plot that any δ_1 values in the range of $[10, 50]$ are roughly equivalent in terms of classification performance. Specifically, for the case of $\delta_1 = 30$ (which is around the range suggested by our guidelines above), Figure 4.2(b) shows the cross-validation scores plotted against different values for δ_3. In this case, as long as δ_3 is in the rough range of $[3, 15]$, the classification performance remains the same.[4]

Furthermore, settings for δ_1 and δ_3 that are safe for classification also select high quality features for debugging. The "smoking gun" predicate that directly indicates the CCRYPT bug is:

```
traverse.c:122:  xreadline() return value == 0
```

This call to `xreadline()` returns 0 if the input terminal is at `EOF`. In all of the above mentioned safe settings for δ_1 and δ_3, this predicate is returned as the top feature.

[4]In Figure 4.2(b), the classification performance for $\delta_1 = 30$ and $\delta_3 = 0$ is deceptively high. In this case, the best δ_2 value is 5, which offsets the cross-validation score by increasing the number of predicted non-crashes, at the expense of worse crash-prediction performance. The top feature becomes a necessary but not sufficient condition for a crash – a false positive-inducing feature! Hence the lesson is that if the bug is believed to be deterministic then δ_3 should always be positive.

The rest of the high ranked features are sufficient, but not necessary, conditions for a crash. The only difference is that, in more optimal settings, the separation between the top feature and the rest can be as large as an order of magnitude; in non-optimal settings (classification score-wise), the separation is smaller.

For BC, the classification results are even less sensitive to the particular settings of δ_1, δ_2, and δ_3. (See Figure 4.2(c,d).) The classification score is roughly constant for $\delta_1 \in [5, 20]$, and for a particular value of δ_1, such as $\delta_1 = 5$, the value of δ_3 has little impact on classification performance. This is to be expected: the bug in BC is non-deterministic, and therefore false positives do indeed exist in the training set. Hence any small value for δ_3 should do.

As for the feature selection results for BC, for all reasonable parameter settings (and even those that do not have the best classification performance), the top features are a group of correlated counters that all point to the index of an array being abnormally big. Below are the top five features for $\delta_1 = 10, \delta_2 = 2, \delta_3 = 1$:

```
1. storage.c:176:  more_arrays():  indx > optopt
2. storage.c:176:  more_arrays():  indx > opterr
3. storage.c:176:  more_arrays():  indx > use_math
4. storage.c:176:  more_arrays():  indx > quiet
5. storage.c:176:  more_arrays():  indx > f_count
```

These features immediately point to line 176 of the file `storage.c`. They also indicate that the variable `indx` seems to be abnormally large. Indeed, `indx` is the array index that runs over the actual array length, which is contained in the integer variable `a_count`. The program may crash long after the first array bound violation, which means that there are many opportunities for the sampling framework to observe the abnormally big value of `indx`. Since there are many comparisons between `indx` and other integer variables, there is a large set of inter-correlated counters, any subset of which may be picked by our algorithm as the top features. In the training run shown

above, the smoking gun of `indx > a_count` is ranked number 8. But in general its rank could be much smaller, because the top features already suffice for predicting crashes and pointing us to the right line in the code.

Chapter 5

The Multi-Bug Problem

The single bug problem is relatively simple. The challenge mainly lies in the existence of non-deterministic bugs, which, as we have shown, can be handled effectively. The problem becomes much more complex in the presence of multiple bugs. We find that the single-bug algorithm becomes ineffective in the multi-bug case. Our efforts in reducing the multi-bug problem to a set of simpler single-bug problems have also been unsuccessful. In the following sections, we illustrate key problems in the multi-bug case, and demonstrate why many natural-sounding ideas do not work.

5.1 Univariate Hypothesis Tests

We first illustrate why simple solutions like univariate hypothesis testing may not yield satisfactory results.

Univariate tests are simple feature selection methods that are often used as a preprocessing step. In text processing, for example, people often perform univariate feature selection to discard unimportant words before proceeding to the real analysis task [Yang and Pedersen, 1997]. Pre-filtering the data results in a smaller feature set, which leads to computational savings in later stages.

Table 5.1: MOSS predicate rankings via two-sample T test on inferred truth probabilities of failures vs. successes.

		Bug Histogram of Runs where Predicate is True							
Rank	Predicate	#1	#2	#3	#4	#5	#6	#7	#9
1	`min_index == lineno`	932	10	12	1	0	2	6	44
2	`min_index == lineno`	985	10	12	1	0	5	7	59
3	`i___0 == lineno`	1085	13	36	1	0	13	13	720
4	`files[filesindex].language > 16`	0	0	25	57	1572	0	0	70
5	`strcmp > 0`	0	0	25	57	1571	0	0	69
6	`tmp___3 == 0 is TRUE`	0	0	25	56	1570	0	0	70
7	`strcmp == 0`	0	0	25	57	1569	0	0	69
8	`min_index > lineno`	1079	18	35	2	0	7	14	685
9	`strcmp > 0`	0	0	25	57	1568	0	0	70
10	`files[filesindex].language == 17`	0	0	25	57	1567	0	0	70
...									

Computational advantages aside, we find that univariate hypothesis tests have limited usefulness for bug-finding. However, in the multi-bug case, the interaction between predicates makes it difficult for univariate algorithms to isolate useful bug predictors.

As a demonstration, let us apply a two-sample T test to the MOSS dataset with non-uniform sampling. We list the top predicates as ranked by the statistic

$$T = \frac{\hat{\pi}_{\mathcal{F}} - \hat{\pi}_{\mathcal{S}}}{\sqrt{\mathrm{Var}_{\mathcal{F},\mathcal{S}}}},$$

where $\mathcal{F}$ and $\mathcal{S}$ respectively denote the set of failing and successful runs, and $\pi_{\mathcal{F}}$ and $\pi_{\mathcal{S}}$ are the average inferred truth probabilities of the predicate in failed and successful runs, respectively. (See chapter 7 for details about how the inferred truth probability is computed.) The denominator is a variance term for the two class: $\mathrm{Var}_{\mathcal{F},\mathcal{S}} = \sigma^2_{\mathcal{F}}/|\mathcal{F}| + \sigma^2_{\mathcal{S}}/|\mathcal{S}|$.

The left hand portion of Table 5.1 lists the top-ranked predicates. The right hand portion contains bug histograms. A bug histogram for a predicate shows the counts of the number of runs in which the predicate is true, binned by the bug that occurred

in that run.[1] We can compute the bug histogram in MOSS because we have the true identification of the bugs that occurred in each run of our dataset.

The bug histograms in Table 5.1 give us a first peek at the phenomena of predicate redundancy and super-bug and sub-bug predictors. We shall clarify the two concepts in subsequent sections. Predicate redundancy manifests itself through the inclusion of predicates 4, 5, 6, 7, 9, and 10 in the ranked list; these are all equally good predictors of bug #5. They directly reveal the fact that, in these failing runs, the input is a Lisp program. Indeed bug #5 lies on the Lisp processing path, and is separated from the rest of the bugs in MOSS. Bug #5 predictors are often the easiest ones for an analysis algorithm to distinguish.

The rest of the top-ranked predicates are not so useful for bug-finding. Predicates 1 and 2 are sub-bug predictors for bug #1. Predicates 3 and 8 are super-bug predictors for bugs #1 and #9.

Based on these results, it appears that simple univariate tests cannot yield succinct bug-predictor lists. Table 5.1 contains a few good predictors for bug #5. But it is difficult to separate the wheat from the chaff without manually examining each predicate – an arduous task that would be too labor-intensive for the users of our system.

5.2 Super-Bug and Sub-Bug Predictors

Our next example again demonstrates that the multi-bug problem cannot be solved by naively applying the single-bug solution. It also reveals the problem of what we call super-bug and sub-bug predictors. Table 5.2 lists the top ten predicates selected by the single-bug algorithm. All the predicates starting with `i > ...` are what we

[1] In the MOSS experiments, multiple bugs could occur in the same failing run. Hence the sum of the histogram counts may sum to a number larger than the total number of failing runs.

Table 5.2: Results of applying the single-bug algorithm on MOSS. ($\delta_1 = 1, \delta_2 = 1, \delta_3 = 0$)

Rank	Function	Predicate
1	`process_file_pass1()`	`(p + passage_index)->last_line < 4`
2	`process_file_pass1()`	`(p + passage_index)->first_line < i`
3	`handle_options()`	`i > 20`
4	`handle_options()`	`i > 26`
5	`process_file_pass1()`	`(p + passage_index)->last_line < i`
6	`handle_options()`	`i > 23`
7	`process_file_pass1()`	`(p + passage_index)->last_line == next`
8	`handle_options()`	`i > 22`
9	`handle_options()`	`i > 25`
10	`handle_options()`	`i > 28`

call *super-bug* predictors, and the rest of the predicates are all *sub-bug* predictors for bug number 1. Neither set is helpful in locating the bugs.

A *super-bug predictor* is a predicate that is mildly correlated with many failed runs, but does not predict any of them very well. All the `i > ...` predicates in Table 5.2 are super-bugs. The variable `i` is the length of the command-line. As a side-effect of our data generation process, MOSS runs with a longer command-line have a higher probability of containing a bug. But there are also many successful runs with a long command-line. A super-bug predictor is often true in many failed runs as well as successful runs. The sheer number of failed runs in which they occur can strengthen their failure prediction power, making them appear better than they are.

A *sub-bug predictor*, on the other hand, is a predicate that is highly correlated with a small subset of failed runs with the same bug. For instance, the `(p + passage_index) ...` predicates in Table 5.2 each predict a small subset of bug 1. Sub-bug predictors hone in on specific failure modes within a certain bug, but are not helpful in locating the actual bug itself.

When we apply the single-bug algorithm to a program containing multiple complex

bugs, we tend to see a mixture of super-bug and sub-bug predictors. The former count for a large number of failed runs, and the latter takes care of leftover small subsets of bugs.

The problem of super-bug and sub-bug predictors can be thought of in terms of a trade-off between *sensitivity* and *specificity*.[2] Let $\mathcal{F}_i$ denote the set of failed runs in which predicate i is observed and true, and $\mathcal{F}$ the entire set of failed runs. Let $\bar{\mathcal{S}}_i$ denote the set of successful runs where predicate i is observed to be false, and $\mathcal{S}$ the entire set of successful runs. The sensitivity of a predicate is defined as the true positive rate, while specificity is the true negative rate:

$$Sens(i) = \frac{|\mathcal{F}_i|}{|\mathcal{F}|} \tag{5.1}$$

$$Spec(i) = \frac{|\bar{\mathcal{S}}_i|}{|\mathcal{S}|}. \tag{5.2}$$

Sensitivity measures how well a predicate covers the set of failed runs. Specificity measures how well its complement covers successful runs. Super-bug predictors have high sensitivity but low specificity. Sub-bug predictors have high specificity but low sensitivity. What we want are predictors with both high sensitivity and specificity.

5.3 Redundancy

Our second problem in the multi-bug case is redundancy. We've already witnessed redundancy at work in the single bug results in section 4.7. The top predicates for BC all contain essentially the same information: the variable `indx` at line 176 of `storage.c` is unusually large. The "smoking gun" of that bug is the predicate `indx > a_count`, which is ranked at number 8. Given the smoking gun predicate,

[2] Sensitivity and specificity are analogous to concepts of recall and precision commonly used in information retrieval.

all the other predicates are redundant, because they do not contain more information than what is already given.

The problem becomes more pronounced in the presence of multiple bugs. In Table 5.1, predicates 4, 5, 6, 7, 9, and 10 all indicate that the input language is Lisp. The strengths of these predictors are all roughly equal. Furthermore, the bug histograms shows that, with the exception of a few runs, these predicates are all either all true or all false.

Even the sub-bug and super-bug predictors in Table 5.2 form redundant sets. Hence redundancy is a separate issue from super-bug and sub-bug predictors. The entanglement of these two problems makes them much more difficult to solve in the case of multiple bugs.

We attempt to resolve the redundancy issue through the following heuristic: given a set of mutually-redundant predicates, we pick the strongest bug predictor to present to the user. If the predictors are all equally strong, then we would pick one at random. Assuming we have a good metric for the strength of a predictor, the problem is to group the predicates into mutually non-redundant sets that would correspond to distinct bugs. We can then pick one representative out of each set as the main bug predictor. By this line of reasoning, it seems that predicate clustering is a natural approach to solving the problem of redundancy.

5.3.1 A Co-Occurrence Similarity Measure

First, we need a similarity measure for predicates. Intuitively speaking, predicates that behave similarly in the same runs should fall into one cluster. Thus we choose to measure predicate similarity via correlation coefficients computed using co-occurrence statistics.

It is often the case that the actual value of the predicate count is not as important

as the fact that it is true at all. For example, when there is an array overrun, we do not need to know whether the predicate `index > array_length` is true five times or fifty times; the fact that the predicate is true already indicates the presence of a bug. Hence from now on we assume that, for the types of bugs that we consider, it is safe to first binarize the predicate counts before computing the co-occurrence statistics.

We also need to deal with the artifacts of sampling. If sampling is sparse, then the predicate may not even be observed in many runs. Unobserved predicates have a count of zero by default. We need to account for this fact by conditioning on the observed values when computing the co-occurrence statistics. Otherwise there would be no way to differentiate a true zero count from something that was simply not observed.

Recall that each predicate accounts for one of a few possible scenarios at an instrumentation site. In fact, the set of predicates at each instrumentation site partitions the outcome space. For example, an integer function return value can be either less than, equal to, or greater than zero. Each site sample increases the truth count of exactly one predicate at that site. Therefore, the sum of these predicate counts gives us the number of times the site is sampled. Thus a non-observed site would have a combined site count of zero. Conditioning on the site count would take the sampling bias out of the individual predicate counts.

This raises a practical concern with the sparsity of data for pairwise statistics. Suppose two sites are both reached exactly once per run. At a sampling rate of d, one would expect to observe individual sites once in $1/d$ runs, and would expect to observe both sites together once in every $1/d^2$ runs. In practice, the sampling rates may differ for different sites, and there may be multiple opportunities to sample a site in each run. But in general, the predicate reports becomes an order of magnitude more sparse for second-order statistics than for univariate statistics. The number of runs we need may also increase by an order of magnitudes. This may be a serious

limitation for methods that use higher order statistics.

We will re-visit the sparsity issue later. For now, let us continue with the investigation of predicate clustering. The correlation coefficient $\rho(X, Y)$ measures the linear correlation strength between two random variables X and Y.

$$\rho(X, Y) = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}}.$$

The denominator ensures that $-1 \leq \rho(X, Y) \leq 1$. If X and Y are binary random variables, ρ measures how much more likely is X to be true when Y is true, and vice versa.

Given two predicates a and b, let A_0 and B_0 denote binary random variables indicating whether they are observed in a particular run; let A and B represent whether or not they are true. We want to estimate the conditional correlation coefficient:

$$\begin{aligned}\rho(A, B \mid A_0 = 1, B_0 = 1) &= \frac{\text{Cov}(A, B \mid A_0 = 1, B_0 = 1)}{\sqrt{\text{Var}(A \mid A_0 = 1)\text{Var}(B \mid B_0 = 1)}} \\ &= \frac{\mathbb{E}(AB \mid A_0 = 1, B_0 = 1) - \mathbb{E}(A \mid A_0 = 1, B_0 = 1)\mathbb{E}(B \mid A_0 = 1, B_0 = 1)}{\sqrt{(\mathbb{E}(A^2 \mid A_0 = 1) - [\mathbb{E}(A \mid A_0 = 1)]^2)(\mathbb{E}(B^2 \mid B_0 = 1) - [\mathbb{E}(B \mid B_0 = 1)]^2)}}.\end{aligned} \tag{5.3}$$

Let $p_a := P(A = 1 \mid A_0 = 1)$, $p_b := P(B = 1 \mid B_0 = 1)$, $p_{ab} := P(A = 1, B = 1 \mid A_0 = 1, B_0 = 1)$. We assume that A is independent from B_0 given A_0 whenever $A_0 \neq B_0$. When $A_0 = B_0$, $A_0 = 1$ and $B_0 = 1$ are the same event, hence no independence assumption is needed. Equation (5.3) can be rewritten as:

$$\rho(A, B \mid A_0 = 1, B_0 = 1) = \frac{p_{ab} - p_a p_b}{\sqrt{p_a(1 - p_a)p_b(1 - p_b)}}.$$

Thus the correlation coefficient can be efficiently computed from data based on plug-in estimates of p_{ab}, p_a, and p_b.

5.3.2 Spectral Clustering

Clustering algorithms can be roughly divided into two categories. Metric-based clustering algorithms such as K-means often form round clusters in metric space. Other clustering algorithms, such as spectral clustering methods, are good at picking out contiguous chain clusters in the original data space. Our decision about which clustering algorithm to use depends on the shape of clusters we expect to find.

Recall our discussion about the effect of sampling on the sparsity of co-occurrence data. 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 all redundant predicates to have a small observed distance to each other. It is much more probable that we would observe, for instance, a being close to b, and b being close to c, but not a being close to c. In this situation, our clustering algorithm should take transitivity into account, and group a, b, and c in the same cluster.

We choose to use spectral clustering methods (see, e.g., [Chung, 1997]). Under the balls-and-springs interpretation of spectral clustering, the predicates are the balls, and their pairwise connection strength are spring constants. If one ball is picked up, then all the other balls in the same cluster should follow. From this perspective, it is easy to see that spectral clustering respects transitivity.

The following spectral clustering algorithm is described in Ng *et al.* [2002]. Let there be n predicates.

1. Given a pairwise similarity matrix $S \in \mathbb{R}^{n \times n}$, form the affinity matrix $A \in \mathbb{R}^{n \times n}$ defined by $A_{ij} = \exp(-S_{ij}/2\sigma^2)$, where σ is a scale parameter.

2. Form the graph Laplacian matrix $L = D^{-1/2}AD^{-1/2}$, where D is a diagonal matrix containing A's row sums.

3. Let K be the number of desired clusters. Compute $Y \in \mathbb{R}^{n \times K}$, the matrix of the K largest eigenvectors of L.

4. Take the i-th row of Y as the new representation of the i-th predicate. Normalize each row to have unit norm, and cluster them using K-means.

We take S to be the predicate correlation coefficient matrix. Correlation coefficient can be negative, but the affinity matrix A guarantees positive entries. We run spectral clustering with K ranging from 5 to 20 and repeat K-means clustering in step 4 ten times for each setting of K, each time starting with random initial conditions. We pick the clustering with the smallest average intra-cluster distance. Note that in this particular clustering algorithm, the dimension of data (the number of eigenvectors used) increases as K increases. Hence a larger K would not necessarily correspond to a smaller average intra-cluster distance.

The model selection process determines the best K to be 9. Figure 5.1 contains bar graphs of the average bug histograms for this clustering. For each cluster, we take the predicates in that cluster and tally the bugs that occur in the runs in which the predicate is true. The resulting bug histogram is then averaged over the number of predicates in each cluster.

An ideal set of bug histograms would contain exactly one peak for each of the bugs. The bug histograms in Figure 5.1 are not very clean. Most of them do not contain a single predominant peak. The exception is cluster 4, which contains most of the predictors for bug #5. Cluster 3 contains some of the predictors for bug #3, but the rest fall under cluster 5 together with a few predictors for bug #1. Predictors for bug #4 are mixed together with predictors for bugs #1 and #9 in cluster 7. The rest of the clusters are mixtures of the rest of the bugs.

The predicate clustering outcome does not seem very promising. Investigation of the clusters reveal that part of the problem stems from what we have called super-bug

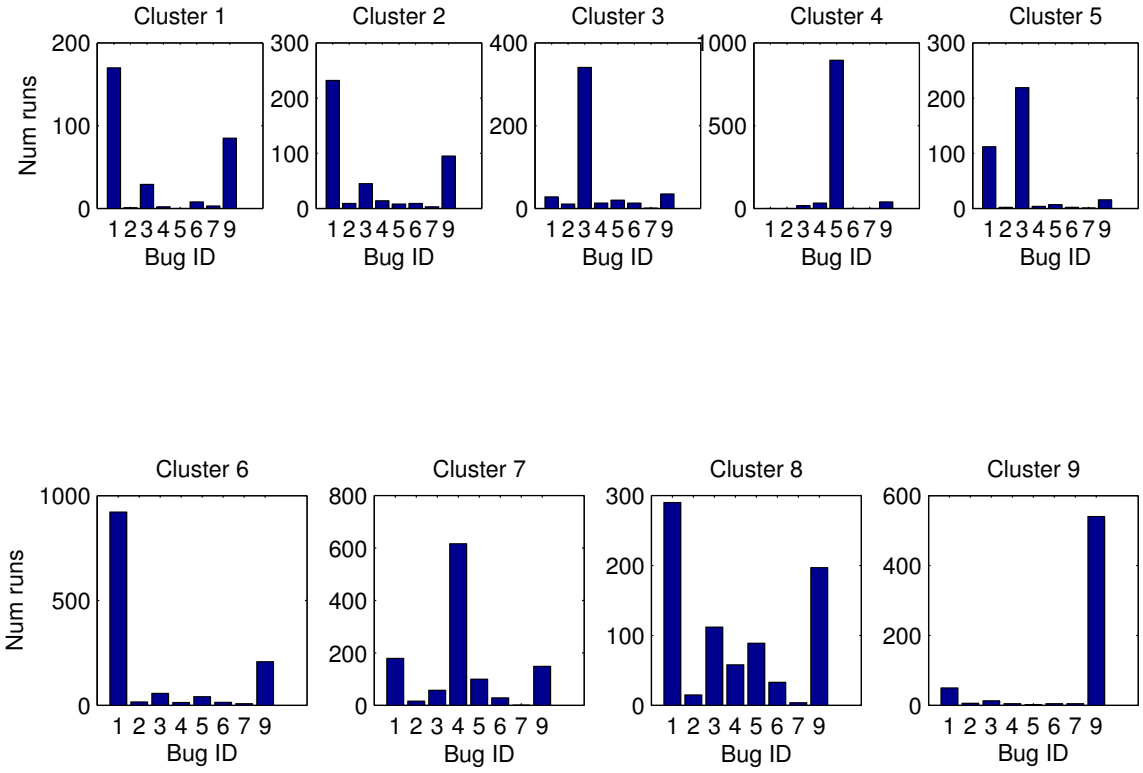

Figure 5.1: Average bug histograms of MOSS predicate clusters.

and sub-bug predictors. A super-bug predictor acts as a liaison between two or more bugs, thus smoothing out the cluster bug histograms. This effect is independent of the particular clustering algorithm we choose. In MOSS, there are some prominent super-bug predictors for bug #1 and bug #9. We have seen examples of these predictors in Table 5.1, and will see a few other examples in section 5.2. These super-bug predictors cause the true predictors of these two bugs to be mixed together and split across many of the predicate clusters.

5.4 Missing Correspondence

Up to now we have discussed the problems of super-bug/sub-bug predictors and redundancy. The former is particular to the multi-bug case, but the latter is a problem for both single-bug and multi-bug programs. The fundamental problem in the multi-bug case is the *missing correspondence* problem between the failed runs and the underlying bug. Suppose there are K bugs in the program. If each failed run could be labeled with a number from 1 to K,[3] then each class of runs can be analyzed separately, which would avoid the problem with super-bug and sub-bug predictors. If one could separately compare each failure class to the set of successful runs, the multi-bug problem would reduce to K single-bug problems.

Unfortunately, we do not have accurate bug labels for all runs. There are a few pieces of potentially useful programmatic information for identifying the bugs: program exit code, fatal signal, stack trace and/or crash site. None of them are reliable enough for bug identification. The exit code is not reliable because, for example, C programmers often default to calling `exit(1)` when there is an unknown error. The fatal signal is not reliable because a single bug may be associated with multiple signals.

[3]More generally, since multiple bugs could occur in the same run, the label could be a set of numbers. In that case, a multi-labeled run would count as an instance of each of the bugs it contains.

In the case of a non-deterministic memory corruption bug, for example, the program may fail at different points in each run, throwing out different fatal signals. The stack trace and/or crash site information could be useful on occasion, but throughout our experimentation we have observed cases where different bugs lead to the same stack trace. Lastly, sometimes the program does not even crash, but still behaves incorrectly. In summary, commonly available debugging information is very coarse. Even combining all of the above information, each distinct evidence vector still would not uniquely identify a bug.

The Missing Correspondence problem appears to be another natural application for clustering. We test this intuition by applying K-means to the MOSS dataset.

Each run is represented by a vector of (non-binary) predicate counts. We include only predicates with non-zero variance. Let x_{ij} denote the count of predicate i in run j, we center and normalize each predicate dimension by subtracting its mean and dividing by its standard deviation:

$$\tilde{x}_{ij} = \frac{x_{ij} - \bar{x}_i}{\bar{\sigma}_i},$$

where $\bar{x}_i$ and $\bar{\sigma}_i$ are the empirical mean and standard deviation, respectively.

We test the clustering algorithm under the ideal situation where the true number of clusters is known and equals the true number of underlying bugs. We repeat K-means several times with random initialization, and pick the clustering with the smallest intra-cluster distance.

Figure 5.2 contains the results. Some of the clusters clearly focus on individual bugs. In particular, clusters 1, 3 and 4 capture bugs #3, #6, and #5, respectively. However, the rest of the clusters are much less distinct. Cluster 2 contains only a small subset of bug #9; cluster 5 contains most of bugs #2 and #4, as well as chunks of bugs #1 and #9. Overall, some clusters contain multiple bugs, while others contain

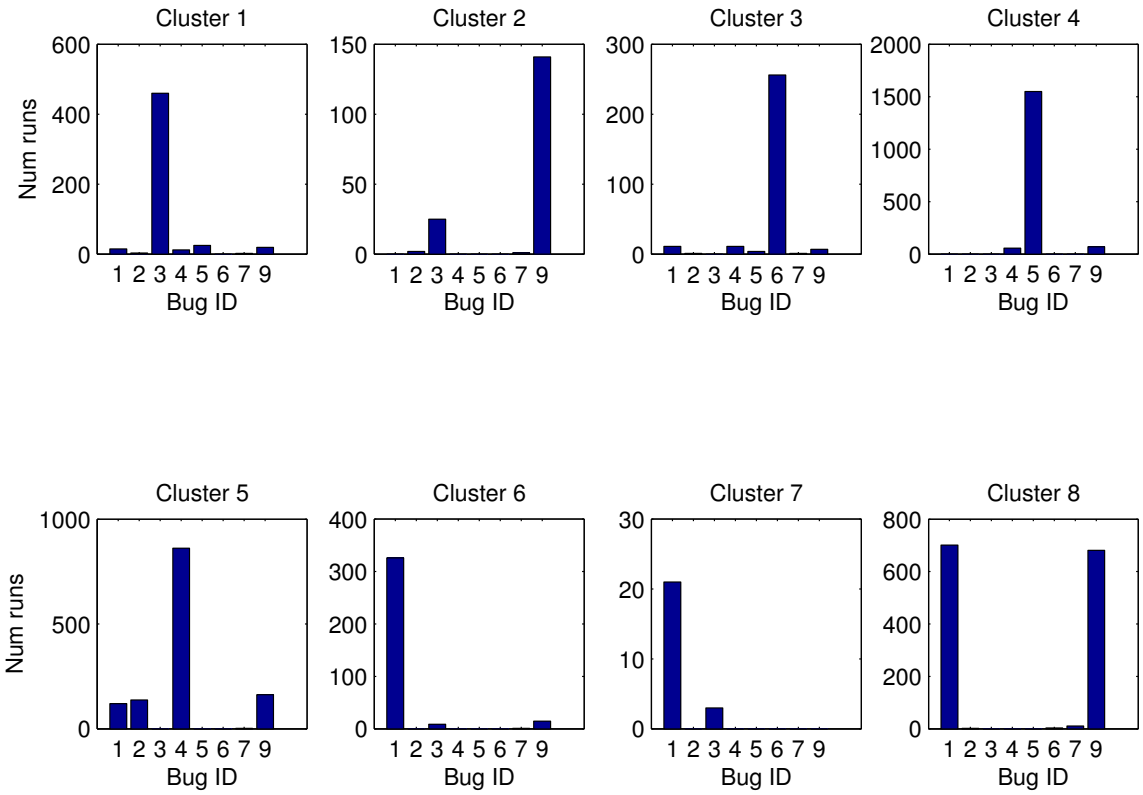

Figure 5.2: Bug histograms of run clusters returned by K-means.

subsets of a single bug. Certain bugs seem prone to be scattered across K-means clusters.

Out of all the MOSS bugs, the Lisp bug (bug #5) is the most easy to isolate, because it lies on a unique branch of the program separate from all other bugs. Bugs #4 and #9 are array overrun bugs that could potentially crash at any point during execution. This may explain why bug #9 is scattered across clusters. The C comment bug (bug #1) is particularly difficult to cluster; it does not cause a failure at all, but produces incorrect output.

A closer analysis of the runs in each cluster reveals why naive clustering does not work on predicate counts. It turns out that these clusters capture usage modes of the program as opposed to failure modes. In hindsight, it seems obvious that the outcome of clustering should be decided by the overall trend of usage modes, as opposed to failure modes which form a much smaller part of the program run-time reports. Any reasonable clustering algorithm would focus on patterns found in the dominant view of the data.

5.5 Summary

Our exploration of the multi-bug problem rules out several naive algorithms. Based on our failed attempts, we make the following observation: runs should clustered by their predictors, and bug predictors should be selected and clustered based on the failing runs they predict. This setup is not unlike the bi-clustering problem [Hartigan, 1972; Tibshirani *et al.*, 1999; Madeira and Oliveira, 2004]. In its basic form, bi-clustering algorithms try to simultaneously cluster both the columns and rows of a data matrix. In recent years there has been much work on applying bi-clustering to applications in bioinformatics [Madeira and Oliveira, 2004]. In our application, however, we have the added advantage of domain knowledge about the program itself. Thus we form

specific constraints about how the predicates should be used to cluster runs, and vice versa.

An alternative to the symmetric approach is to attack one side first. For example, we may choose to focus on first obtaining good run clusters. Runs should be clustered based on a few select bug-predictors. This setup is similar to that of the COSA framework for clustering on select subsets of features. Friedman and Meulman [2004] explored several possible formulations of the problem. Finding good clusters on unspecified subsets of attributes is a difficult optimization problem in general. But here we can again use our knowledge about predicate behavior to direct the search process. By defining a set of predicate update equations, we tailor our algorithm toward specific types of bugs. We introduce the multi-bug algorithm in the next chapter.

Chapter 6

The Multi-Bug Algorithm

In the previous chapter, we identified three difficult problems in the case of multiple bugs: super-bug and sub-bug predictors, redundancy of predicates, and missing correspondence between crashes and the underlying bug. In this chapter, we propose a solution based upon the following symmetry principle:

> *Predicates should group by the runs that they predict; runs should group by the predicates that predict them.*

Our multi-bug algorithm aims to find the right balance between predicate clusters and run clusters, which would in turn lead to the right balance between sensitivity and specificity of the selected predicates.

We start with the pre-filtering stage of our multi-bug algorithm in section 6.1. While the univariate predcate filter has meaningful program analysis motivations, its main purpose is to discard uninteresting predicates and lighten the computation burden on subsequent stages of the algorithm. In section 6.2, we lay out the criteria for a good bug predictor that incorporate the notions of sensitivity and specificity.

The overall goal of our multi-bug algorithm is to identify important predicates that are useful in predicting large sets of failed runs. We define predicate quality and

run attribution functions, and iterate the set of update equations until reaching a fixed point. section 6.4 presents the predicate and run update equations. Predicate quality is defined in accordance with the bug predictor criteria, with an emphasis on balancing the tension between sensitivity and specificity. The run attribution functions allow the runs to cast votes for each predicate. The amount of vote received by a predicate depends on the quality of the predicate itself, and is also influenced by the quality of other predicates competing for the vote. This setup encodes the notion of redundancy, and attempts to minimize its effect through mutual competition amongst the predicates.

The update equations are iterated. After a fixed point is reached, the runs go through an additional round of voting, this time settling on only one predicate. We conclude section 6.4 with an analogy to the disjunction-learning problem.

The predicate and run updates are iterated on a complete bipartite graph. Overall we have found it more handy to represent this algorithm using a non-probabilistic graphical model rather than a probabilistic one. In section 6.3, we go into more details about our modeling choice, and explain why the problem is not naturally amenable to, say, a QMR-DT-like noisy-or model.

We conclude the chapter with a summary of the algorithm in section 6.5.

6.1 Predicate Pre-Filter

Our first step is a predicate pre-filter. The filter retains only a small subset of potentially interesting predicates, thereby easing the computational burden for later stages of the algorithm. We motivate the predicate pre-filter from a program analysis perspective. As it turns out, the filtering criterion is also akin to a conservative likelihood ratio test.

Consider the following C code fragment:

```
f = ...;           (a)
if (f == NULL) {   (b)
    x = 0;         (c)
    *f;            (d)
}
```

The branch-conditional predicate `f == NULL` on line `(b)` is clearly highly correlated with failure. In fact, whenever it is true this program inevitably crashes.[1] In general, it is useful to look at the probability of crashing when a predicate is true:

$$\hat{P}(Crash \mid \text{predicate } i \text{ is true}) = \frac{|\mathcal{F}^i|}{|\mathcal{F}^i| + |\mathcal{S}^i|},$$

where $\mathcal{F}^i$ and $\mathcal{S}^i$ respectively denote the sets of failed and successful runs in which predicate i is true.

However, this is not the only probability of interest. Take a look at line `(c)` above. The scalar-comparison predicate `x == 0` on line `(c)` is also true whenever the program crashes. But it has nothing to do with the bug. It is just an "innocent bystander" on the path of failure. In fact, even if the predicate is false, the program would still crash. The fact that it is even observed already spells trouble for the program. Hence we postulate that the following probability is also of interest:

$$\hat{P}(Crash \mid \text{predicate } i \text{ is observed}) = \frac{|\mathcal{F}^i_{obs}|}{|\mathcal{F}^i_{obs}| + |\mathcal{S}^i_{obs}|},$$

where $\mathcal{F}^i_{obs}$ and $\mathcal{S}^i_{obs}$ respectively denote the sets of failed and successful runs in which predicate i is observed.

Our code example illustrates a particular case of cause-effect redundancy. Redundancy occurs because many predicates are correlated with the same bug. In some

[1] The converse is not true in multi-bug programs even for the best bug predictors: it is not the case that a bug predictor would be true in all crashed runs.

cases, such correlation is a manifestation of cause and effect. Often times, all predicates that lie down stream from the first major indication of failure can be interpreted as effects of the bug. It may be possible to construct cause-effect chains via static analysis of the program source code. But such cause-effect information is often imperfect and incomplete. Furthermore, predicate chain cause-effect graphs would be of staggering size and complexity. Due to these considerations, we have chosen not to take the causal analysis approach in this project.

However, it is possible to deal with the cause-effect confusion and eliminate some correlated predicates in certain code patterns. Specifically, when a "wrong" decision (e.g., `f == NULL` in our example) leads the program down an erroneous path, we can separate the initial decision from the subsequent innocent bystander predicates.

Based on observations from our example, we say that predicate i is interesting if and only if

$$\hat{P}(Crash \mid \text{predicate } i \text{ is true}) > \hat{P}(Crash \mid \text{predicate } i \text{ is observed}). \tag{6.1}$$

This condition is equivalent to a simple hypothesis test. Consider the two classes of runs: failed runs $\mathcal{F}$ and successful runs $\mathcal{S}$. For each class, we can treat predicate i as a Bernoulli random variable with heads probabilities π_f^i and π_s^i, respectively. We take π to be the probability that the predicate is true, conditioned on the fact that it is observed. π_f^i and π_s^i can be estimated as:

$$\hat{\pi}_f^i = \frac{|\mathcal{F}^i|}{|\mathcal{F}_{obs}^i|} \qquad\qquad \hat{\pi}_s^i = \frac{|\mathcal{S}^i|}{|\mathcal{S}_{obs}^i|}.$$

If a predicate is indeed a bug predictor, then π_f^i should be bigger than π_s^i. Our null hypothesis is $\mathcal{H}_0 : \pi_f^i \leq \pi_s^i$, and the alternate hypothesis is $\mathcal{H}_1 : \pi_f^i > \pi_s^i$. We calculate

the two-sample mean T test statistic:

$$T = \frac{\hat{\pi}_f^i - \pi_s^i}{V_{f,s}^i},$$

where $V_{f,s}^i$ is a sample variance term [Lehmann, 1986]. A necessary (but not sufficient) condition for rejecting the null hypothesis is that $\hat{\pi}_f^i > \hat{\pi}_s^i$. This is equivalent to the condition in Equation (6.1).

Lemma 1. *The condition*

$$P(Crash \mid \textit{predicate } i \textit{ is true}) > P(Crash \mid \textit{predicate } i \textit{ is observed})$$

is equivalent to the naive test $\hat{\pi}_f^i > \hat{\pi}_s^i$.

Proof. For simplicity, let $a = |\mathcal{F}^i|$, $b = |\mathcal{S}^i|$, $c = |\mathcal{F}_{obs}^i|$, and $d = |\mathcal{S}_{obs}^i|$. We have:

$$\begin{aligned}
&\hat{P}(Crash \mid \text{predicate } i \text{ is true}) > P(Crash \mid \text{predicate } i \text{ is observed}) \\
\iff &\frac{a}{a+b} > \frac{c}{c+d} \\
\iff &a(c+d) > (a+b)c \\
\iff &ad > bc \\
\iff &\frac{a}{c} > \frac{b}{d} \\
\iff &\hat{\pi}_f^i > \hat{\pi}_s^i. \quad \square
\end{aligned}$$

6.2 Definition of Bug Prediction Strength

In order to define predicate quality, we need a good definition for the bug prediction strength of a predicate. A bug predictor should be something that is true when a

failure occurs.[2] But there are a few additional requirements that are also reasonable.

Recall our code fragment example:

```
f = ...;           (a)
if (f == NULL) {   (b)
    x = 0;         (c)
    *f;            (d)
}
```

The predicate `f == NULL` on line `(b)` is the correct bug predictor. If the program fails due to this bug, then `f == NULL` must be true; if the program makes a successful exit, then either its complement predicate `f != NULL` is true, or the program simply never traverses this path and neither `f == NULL` nor its complement is observed. The example illustrates a common phenomenon. Hence we say that a good bug predictor is a predicate that is true whenever the program fails and is not true when the program succeeds.[3] Its complement should always be true when the program succeeds, and should not be true when the program fails.

Definition 1. Let P be a predicate, and let $\bar{P}$ be the logical complement of P (e.g., if P denotes the predicate `x > y`, then $\bar{P}$ denotes `x <= y`). If P is a good *bug predictor*, then

- P should be true in many failed runs;

[2] For computational purposes we must make certain modeling assumptions. But there is no universally true assumption that works for all types of bugs. For instance, certain types of bugs violate the "true-when-fail" assumption here. Forgotten initialization bugs occur when programmers forget to initialize variables before using them. Shared memory race conditions occur if one of the program threads forgets to lock the area before attempting access. In both cases, the relevant predicates would be false.

[3] The criteria listed here work for a deterministic bug, but may be violated by a non-deterministic one. Even in the latter case, it may be enlightening to see the most deterministic predictor. In subsequent sections, we will see that these criteria are used not as hard constraints, but rather as penalization. Our experimental results in chapter 8 confirm that this definition works for non-deterministic bugs.

- P should not be true in many successful runs;
- $\bar{P}$ should be true in many successful runs;
- $\bar{P}$ should not be true in many failed runs.

These conditions are not redundant of each other. An instrumentation site may be reached multiple times during a single run of the program. Hence P and $\bar{P}$ may both be true in a run. In fact, this happens very frequently. Under 1, the bug-prediction strengths of P and $\bar{P}$ would work against each other.

In previous work [Liblit *et al.*, 2005], we found that the harmonic mean between sensitivity and specificity is helpful in resolving the confusion between super-bug and sub-bug predicates. Recall our definition of sensitivity and specificity in Equation (5.1) and Equation (5.2). The measurement of sensitivity depends on both the true probability and the observation probability of a predicate. Specificity, on the other hand, depends on the truth probability and observatino probability of the complement of the predicate. Our definition above directly incorporates the concepts of sensitivity and specificity by accounting for the behavior of both the predicate itself and its complement. In section 6.4, we present predicate update equations that attempts to balance the tension between sensitivity and specificity, much like a harmonic mean.

6.3 Graphical Representation of the Model

Given the interlocking relationship between predicates and runs, it is natural to model our problem using a bipartite graph. Figure 6.1 contains a bipartite graphical representation of our model. The top row contains n runs, colored and labeled with their exit status of failure or success. The bottom row lists m predicates. Every predicate is connected to every run. The connection strengths are represented by the affinity

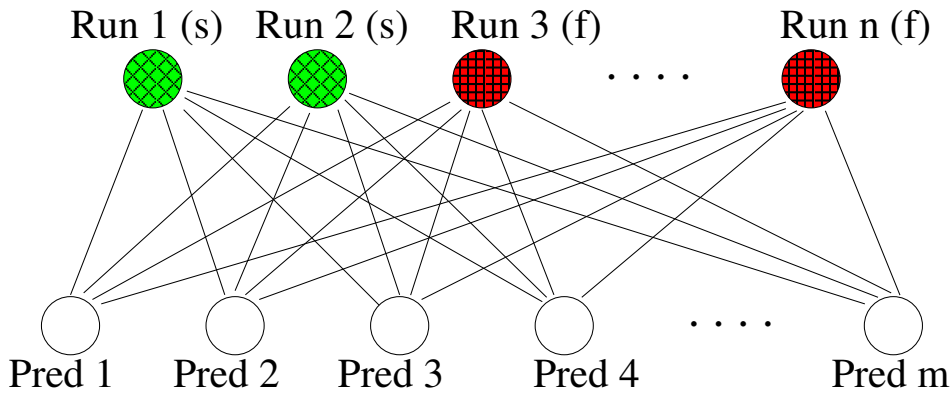

Figure 6.1: A model for the multi-bug constraint-satisfaction model. Runs in the top row are labeled and colored according to their exit status of either failure or success.

matrix $A \in \mathbb{R}^{m \times n}$. For now we assume that all entries in A are either 0 or 1, where $A_{ij} = 1$ if and only if predicate i is true in run j. In chapter 7, we discuss how to extend these binary weights to real-valued weights.

Note that Figure 6.1 is not a probabilistic graphical model. At first glance, it may look like an upside-down and undirected version of the QMR-DT network [Jaakkola and Jordan, 1999]. But the noisy-or model is not suitable for our problem.

Figure 6.2 contains a noisy-or representation of the debugging problem. $P_1, \ldots, P_m$ represent predicate counts and R represents the run outcome, all of which are observed. Each P_i is associated with an inhibition probability Z_i, whose value is not known and must be learned. S specifies the parameters of a prior distribution on the Z_i's. The joint probability may be rewritten as:

$$P(s, \mathbf{p}, \mathbf{z}, r) = P(s)P(\mathbf{z} \mid s) \prod_{i=1}^{m} P(p_i)P(r \mid \mathbf{p}, \mathbf{z}),$$

where

$$P(r = 0 \mid \mathbf{p}, \mathbf{z}) = \prod_{i=1}^{m} (1 - z_i)^{p_i}.$$

Our goal is to use as few predicates as possible in predicting the outcome of a run.

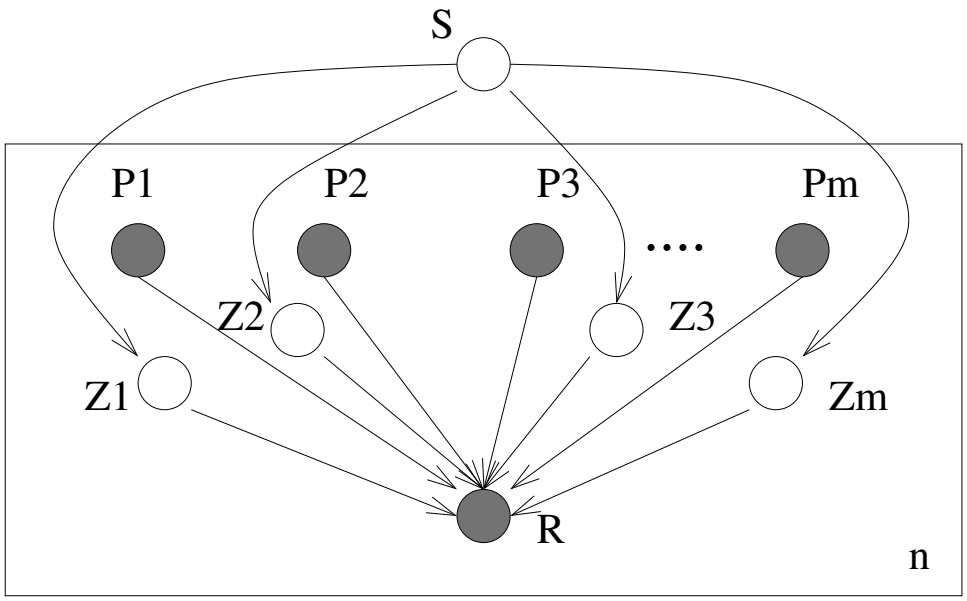

Figure 6.2: A noisy-or representation of the multi-bug problem.

In the noisy-or model, this is equivalent to setting the values of Z_i so that only a few of them have non-zero values. In order to achieve this, $P(\mathbf{z} \mid s)$ would have to act as a sparsifying prior. For each run, then, inference based on observed evidence would activate only a few Z_i's. It is far from clear how such a prior should be specified so as to obtain the desired effects. Overall, the constraint satisfaction framework built upon a non-probabilistic bipartite graph allows for a much more natural and direct predicate selection process.

6.4 Predicate and Run Update Equations

We are now ready to define the predicate and run update equations. Let Q_i and $Q_{\bar{i}}$ denote the quality of predicate i and its complement, respectively:

$$\begin{aligned} Q_i &= \frac{F_i}{S_i} \cdot \frac{S_{\bar{i}}}{F_{\bar{i}}}, \\ Q_{\bar{i}} &= \frac{1}{Q_i}. \end{aligned}$$

The precise mathematical definitions of F_i, $F_{\bar{i}}$, S_i, and $S_{\bar{i}}$ will be given later on in this section. F_i represents the contribution of predicate i to the set of failed runs, and $F_{\bar{i}}$ the contribution of the complement of predicate i to the set of failed runs. S_i and $S_{\bar{i}}$ have analogous interpretations. In accordance with Definition 1, Q_i is large only when the predicate contributes more to failing runs than successful runs and its complement contributes more to successful runs than failing runs.

Our algorithm is essentially a collective voting process. The quality of a predicate depends on the amount of votes it receives from the runs; the amount of vote given by a run in turn depends on the quality of the predicate and all of its competitors. Thus the definitions of F_i, S_i, $F_{\bar{i}}$ and $S_{\bar{i}}$ involves the following three quantities: the predicate-run connection weight A_{ij}, the predicate-run contribution R_{ij}, and the residual weight of a run.

A_{ij}: Each predicate and run connection in our graph is associated with an edge weight. The weights can be binary indicators of whether or not the predicate was true in that run, or it can be a posterior predicate-run truth probability. Either way, they can be interpreted as an objective measurement of "qualification." If the weight is zero, then the predicate should have no effect on this run. Otherwise the effect should be proportional to the connection weight.

R_{ij}: R_{ij} is the residual contribution of predicate i to run j. It is defined using the predicate-run connection weight A_{ij} and the predicate quality score Q_i. Recall that the larger Q_i is, the more effect predicate i has over failing runs, and the less effect it has over successful runs. Thus, R_{ij} should be directly proportional to Q_i in failing runs, and inversely proportional to Q_i in successful runs.

$$R_{ij} = \begin{cases} A_{ij}Q_i, \text{ if } j \in \mathcal{F} \\ A_{ij}/Q_i, \text{ if } j \in \mathcal{S} \end{cases}$$

where $\mathcal{F}$ denotes the set of failed runs, and $\mathcal{S}$ the set of successful runs. Similarly, we define the residual contribution of the complement of predicate i to run j:

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

Run residual weight: Recall the redundancy problem discussed in the previous chapter. Each run can be predicted by many predicates. In order to reduce redundancy, we need to set up the update equations so that predicates compete with each other for the coverage of each run. If a run has already been accounted for by other predicates, then there should not be much left for the current predicate to predict. We set the total weight of each run to 1, and define the residual weight of run j for predicate i to be $(1 - \sum_{k \neq i} R_{kj})$. This definition ensures that predicate i can only account for the leftover portion of run j that has not yet been accounted for by any other predicate.

We are now ready to give the definitions of F_i, $F_{\bar{i}}$, S_i, and $S_{\bar{i}}$. We take the contribution of a predicate to a set of runs to be the sum of its contribution to each individual run. The contribution of a predicate to a single run is directly proportional

to the run residual weight and the predicate-run connection weight. Thus F_i is defined as:

$$F_i = \sum_{j \in \mathcal{F}} A_{ij}(1 - \sum_{k \neq i} R_{kj}).$$

Similarly, $F_{\bar{i}}$ is defined as:

$$F_{\bar{i}} = \sum_{j \in \mathcal{F}} A_{\bar{i}j}(1 - \sum_{k \neq i} R_{\bar{k}j}).$$

S_i and $S_{\bar{i}}$ are defined analogously.

The multi-bug problem is closely related to the problem of disjunction learning. The update equations can be interpreted through an analogy to set-covering machines. A set-covering machine [Marchand and Shawe-Taylor, 2002; Haussler, 1988; Valiant, 1984] tries to learn a sparse disjunction or conjunction of a set of Boolean-valued features based on positive and negative examples. The machine starts out with the set of features that are consistent with all the negative examples, i.e., features that are never true whenever the outcome is zero. This set is then reduced to a sparse set of features that cover all positive examples. More generally, the merit of a feature is defined as a trade-off between its influence on positive and negative examples. The learning process alternates between computing the feature merit function based on the positive examples that have not yet been covered, and selecting the best feature left in the candidate set.

In our case, the failed runs are positive examples of a bug, and successful runs negative examples. The predicates are the features that must cover the set of failed runs while being consistent with the successful runs. The merit of a predicate can be defined iteratively, based on the set of runs it covers and whether there are other predicates competing for that cover.

6.5 The Algorithm

The multi-bug algorithm is summarized in Algorithm 1. The first stage (lines 2–4) is the predicate pre-filtering step. We next initialize necessary quantities in lines 7–10. The third stage (lines 13–22) iterates the predicate-run update equations until convergence. The fourth stage (lines 25–27) assigns each failed run to the predicate that makes the most contribution, where the single predicate contribution is weighted by the contributions towards the successful runs. The final stage (line 30) ranks the predicates by the number of failed runs they account for.

The algorithm is essentially a collective voting process. Each run has one vote to cast for which predicate it prefers. At the beginning, all predicates are equal, and the runs can give fractions of their votes to each of the candidates, depending on how attractive they seem. In this stage of the algorithm, the runs can influence each other's vote distributions through the predicate update equations. Popular candidates receive even more votes. After this process converges, each run must now cast the entire vote toward their favorite candidate. The predicates are then ranked by the number of votes they receive.

The iterative updates are designed to find a natural equilibrium between sensitivity and specificity. A highly specific predicate would have a high contribution towards certain runs, whereas a sensitive predicate would have some contribution towards a large set of runs. The strongest predicates would naturally enjoy both advantages, thus beating its competition in the voting game.

Note that this algorithm makes no explicit use of high order statistics other than the run-predicate connection weight A_{ij}. The score of each predicate influences the scores of all other predicates through the update equations, but there is no direct interaction between predicates. The sufficient statistics required by the algorithm (including those for the truth-value model described in chapter 7) can be collected

Algorithm 1 Algorithm for finding bug predictors in the presence of multiple bugs.

// **Pre-filter predicates**
for all predicate i **do**
  retain predicate i iff

$$P(Crash \mid \text{predicate } i \text{ is true}) > P(Crash \mid \text{predicate } i \text{ is observed})$$

end for

// **Initialize**
Set ϵ to a small number, say $1e^{-12}$
for all retained predicate i **do**
  $Q_i^0 \Leftarrow 1$, $Q_{\bar{i}}^0 \Leftarrow 1$, $t \Leftarrow 0$
end for

// **Run updates**
repeat
  for all retained predicate i **do**
    for all run j **do**

$$R_{ij}^t \Leftarrow \begin{cases} A_{ij} Q_i^{t-1}, \text{ if } j \in \mathcal{F} \\ A_{ij} / Q_i^{t-1}, \text{ if } j \in \mathcal{S} \end{cases} \qquad R_{\bar{i}j}^t \Leftarrow \begin{cases} A_{\bar{i}j} Q_{\bar{i}}^{t-1}, \text{ if } j \in \mathcal{F} \\ A_{\bar{i}j} / Q_{\bar{i}}^{t-1}. \text{ if } j \in \mathcal{S} \end{cases}$$

    end for
    $F_i^t \Leftarrow \sum_{j \in \mathcal{F}} A_{ij}(1 - \sum_{k \neq i} R_{kj}^t)$, $F_{\bar{i}}^t \Leftarrow \sum_{j \in \mathcal{F}} A_{\bar{i}j}(1 - \sum_{k \neq i} R_{\bar{k}j}^t)$
    $S_i^t \Leftarrow \sum_{j \in \mathcal{S}} A_{ij}(1 - \sum_{k \neq i} R_{kj}^t)$, $S_{\bar{i}}^t \Leftarrow \sum_{j \in \mathcal{S}} A_{\bar{i}j}(1 - \sum_{k \neq i} R_{\bar{k}j}^t)$
    $Q_i^t \Leftarrow (F_i^t S_{\bar{i}}^t)/(S_i^t F_{\bar{i}}^t)$, $Q_{\bar{i}}^t = 1/Q_i^t$
  end for
until $\sum_i (Q_i^t - Q_i^{t-1})^2 < \epsilon$

// **Cast votes**
for all run $j \in \mathcal{F}$ **do**
  Assign run j to predicate ℓ, where $\ell = \text{argmax}_i \frac{A_{ij}(1-\sum_{k \neq i} R_{kj})}{A_{\bar{i}j}(1-\sum_{k \neq \bar{i}} R_{kj})} \cdot \frac{S_{\bar{i}}}{S_i}$.
end for

// **Rank predicates**
Sort predicates by the number of runs they account for.

individually for each predicate. In this sense Algorithm 1 is a univariate algorithm. Because we no longer need higher-order statistics, we no longer need to worry about the data sparsity issue that had plagued our previous predicate clustering attempt in section 5.3. This is not unlike the justification for belief propagation for approximate inference on general graphical models. We harness the power of iterative updates to simulate direct interaction between predicates and runs, something that would otherwise be expensive to compute.

Chapter 7

Inferring Predicate Truth Probabilities

In this chapter, we focus on computing the predicate-run connection weights A_{ij} in our multi-bug model. Recall our bipartite graphical representation of the problem, reproduced in Figure 7.1 for convenience. We associate a weight A_{ij} with the edge between predicate i and run j. The weights can be as simple as the observed binary predicates truth counts, i.e., $A_{ij} = 1$ iff predicate i is true in run j. However, these binary counts may become overly sparse in sampled data. Sampled data is also biased towards predicates that are reached more often during a run of the program: all else being equal, predicates that are reached more often have a higher chance of being observed to be true.

The predicates truth counts that we observe are but fragments of the underlying reality. The ideal A_{ij} weights are the unobserved real predicate truth counts. But based on the observed counts and what we know about the sampling process, we can infer the probability that the predicate was true in any particular run. We can think of this as *imputing* the predicate-run connection weight that may be missing due to sampling. In doing so, we hope to ameliorate the data sparsity problem, and also correct biasness caused by sampling.

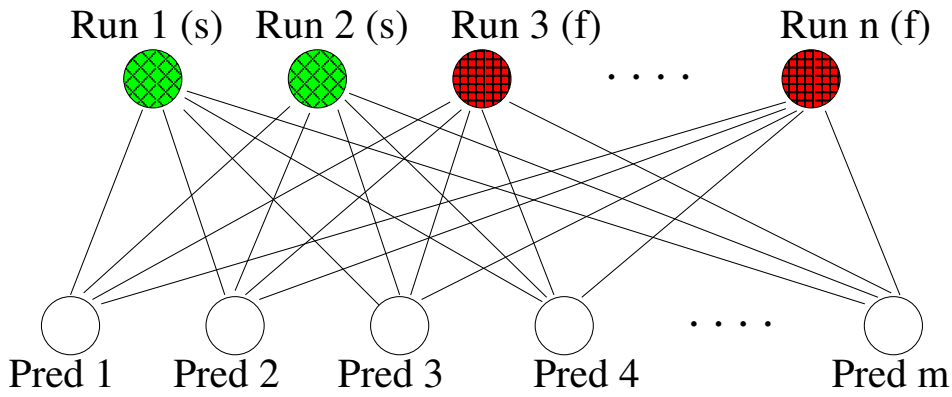

Figure 7.1: A model for the multi-bug constraint-satisfaction model. Runs in the top row are labeled and colored according to their exit status of either failure or success.

7.1 The Truth Probability Model

Figure 7.2 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 found 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, the number of times a site is reached often follows a Poisson distribution. Occasionally, the site may not be reached at all during a certain run. Hence we endow N with a prior that is a mixture of a Poisson distribution and a spike at zero.

One may naively model the conditional probability of X given N as a binomial. This corresponds to the assumption that the predicate truth counts come from a Bernoulli process. While this assumption is not entirely unreasonable, closer examinations of model fitness on the MOSS dataset shows that there is also a "sticky" mode in the truth values. In certain runs, it may be impossible for the predicate to

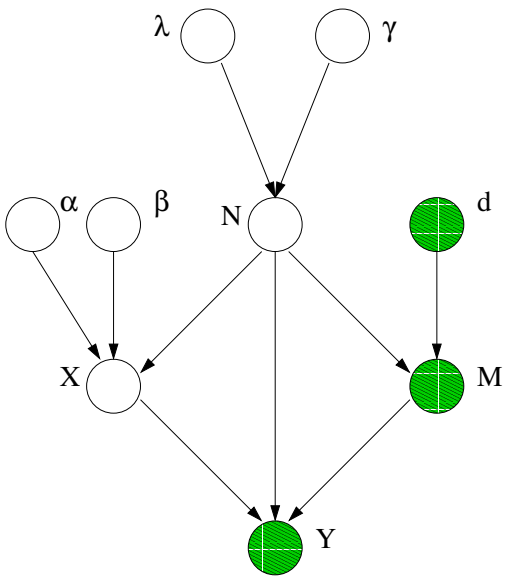

Figure 7.2: Predicate truth probability model.

be true, while in other runs, it may always be true. Hence we model the conditional probability of X given N as a mixture of a binomial distribution, a spike at zero, and a spike at N.

The rest of the conditional probabilities may be directly specified based on our knowledge of the sampling process. $P(M \mid N)$ is a binomial distribution with parameter d, and Y has a hypergeometric distribution given M, N, and X.

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

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

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

We endow the hyperparameters 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.

$$\begin{aligned}
\alpha &\sim \text{Beta}(s, t), \\
\boldsymbol{\beta} &\sim \text{Dir}(c_1, c_2, c_3), \\
\gamma &\sim \text{Beta}(j, k), \\
\lambda &\sim \text{Gam}(u, v).
\end{aligned}$$

The hyperparameters t, s, $\{c_i\}$, j, k, u, and v are set so that the mean and variance of α, $\boldsymbol{\beta}$, γ, and λ equal their empirical means and variances.

$$\begin{aligned}
t &= (\text{total \# truth counts}) + 1 \\
s &= (\text{total \# false counts}) + 1 = (\text{total \# obs}) - (\text{total \# true}) + 1 \\
c_1 &= (\text{\# runs where } m > y > 0) + 1 \\
c_2 &= (\text{\# runs where } m > y = 0) + 1 \\
c_3 &= (\text{\# runs where } y = m > 0) + 1 \\
j &= (\text{\# runs where } m = 0) + 1 \\
k &= (\text{\# runs where } m > 0) + 1 \\
u &= \frac{(\text{average \# obs per run})^2}{(\text{variance of \# obs per run})} \\
v &= \frac{(\text{variance of \# obs per run})}{(\text{average \# obs per run})}
\end{aligned}$$

7.2 MAP Estimates of Hyperparameters

We need to compute either the posterior distribution of the hyperparameters α, $\boldsymbol{\beta}$, γ, and λ, or set them to some specific values following the empirical Bayes methodology. We take the latter approach and set the parameters to their maximum-a-posteriori estimates given the observations. Suppose the dataset contains r runs. Let $\mathbf{m} = \{m_1, \ldots, m_r\}$ and $\mathbf{y} = \{y_1, \ldots, y_r\}$ be the sequence of observed counts and truth counts of a single predicate. The MAP estimates $\hat{\alpha}$, $\hat{\boldsymbol{\beta}}$, $\hat{\gamma}$, and $\hat{\lambda}$ are taken to be:

$$\begin{aligned}
\{\hat{\alpha}, \hat{\boldsymbol{\beta}}, \hat{\lambda}, \hat{\gamma}\} &= \text{argmax}_{\{\alpha,\boldsymbol{\beta},\lambda,\gamma\}} P(\alpha, \boldsymbol{\beta}, \lambda, \gamma \mid \mathbf{m}, \mathbf{y}) \\
&= \text{argmax}_{\{\alpha,\boldsymbol{\beta},\lambda,\gamma\}} P(\alpha) P(\boldsymbol{\beta}) P(\gamma) P(\lambda) P(\mathbf{y} \mid \mathbf{m}, \alpha, \boldsymbol{\beta}) P(\mathbf{m} \mid \lambda, \gamma, d) \\
&= \text{argmax}_{\{\alpha,\boldsymbol{\beta},\lambda,\gamma\}} P(\alpha, \boldsymbol{\beta}, \mathbf{y} \mid \mathbf{m}) P(\lambda, \gamma, \mathbf{m} \mid d), \qquad (7.1)
\end{aligned}$$

where $P(\alpha, \boldsymbol{\beta}, \mathbf{y} \mid \mathbf{m}) := P(\alpha) P(\boldsymbol{\beta}) P(\mathbf{y} \mid \mathbf{m}, \alpha, \boldsymbol{\beta})$, and $P(\lambda, \gamma, \mathbf{m} \mid d) := P(\lambda) P(\gamma) P(\mathbf{m} \mid \gamma, \lambda, d)$.

To obtain the MAP estimates, we maximize Equation (7.1) using Newton-Raphson. Note that the probability is factorizable, which means that we can estimate $\{\hat{\alpha}, \hat{\boldsymbol{\beta}}\}$ separately from $\{\hat{\gamma}, \hat{\lambda}\}$. Appendix A contains detailed calculations of the gradients and Hessians for use in Newton-Raphson. In the remainder of this section, we derive the closed form formulas for $P(\mathbf{y} \mid \mathbf{m}, \alpha, \boldsymbol{\beta})$ and $P(\mathbf{m} \mid \gamma, \lambda, d)$, both of which are needed in Equation (7.1).

To simplify notation, we do not explicitly condition on irrelevant hyperparameters in the following derivations. We start by marginalizing out X. This turns out to affect

only the conditional distribution of Y:

$$
\begin{aligned}
&P(n, m, y \mid \alpha, \boldsymbol{\beta}) = \sum_x P(n, m, x, y \mid \alpha, \boldsymbol{\beta}) \\
&= P(n)P(m \mid n) \\
&\quad \cdot \sum_x \left(\beta_1 \mathrm{Bin}(x; n, \alpha) + \beta_2 \delta(x = 0) + \beta_3 \delta(x = n)\right) \mathrm{Hypergeo}(y \mid m, n, x) \\
&= P(n)P(m \mid n) \cdot \\
&\quad (\beta_1 \sum_{x=y}^{n-(m-y)} \frac{n!}{x!(n-x)!} \frac{x!}{y!(x-y)!} \frac{(n-x)!}{(m-y)!(n-x-(m-y))!} \alpha^x (1-\alpha)^{n-x} \\
&\quad + \beta_2 \delta(y = 0) + \beta_3 \delta(y = m)) \\
&= P(n)P(m \mid n)(\beta_1 \binom{m}{y} \alpha^y (1-\alpha)^{m-y} \sum_{x=y}^{n-(m-y)} \binom{n-m}{x-y} \alpha^{x-y} (1-\alpha)^{n-x-(m-y)} \\
&\qquad\qquad + \beta_2 \delta(y = 0) + \beta_3 \delta(y = m)) \qquad (7.2) \\
&= P(n)P(m \mid n)(\beta_1 \mathrm{Bin}(y; m, \alpha) + \beta_2 \delta(y = 0) + \beta_3 \delta(y = m)) \qquad (7.3) \\
&= P(n)P(m \mid n)P(y \mid m, \alpha, \beta).
\end{aligned}
$$

The term under the summation in Equation (7.2) forms a binomial distribution of $(x - y)$ given $(n - m)$ and α and thus sums to one. The new marginal conditional probability $P(y \mid m, \alpha, \boldsymbol{\beta})$ is a mixture of a binomial distribution and delta functions at 0 and M.

Next we marginalize out N, which induces a distribution on M that is a mixture

of a Poisson and a delta function at zero:

$$
\begin{aligned}
P(m, y \mid \alpha, \boldsymbol{\beta}, \lambda, \gamma, d) &= \sum_{n=m}^{\infty} P(n, m, y \mid \alpha, \boldsymbol{\beta}, \lambda, \gamma, d) \\
&= P(y \mid m, \alpha, \boldsymbol{\beta}) \sum_{n=m}^{\infty} P(n \mid \lambda, \gamma) \mathrm{Bin}(m; n, d) \\
&= P(y \mid m, \alpha, \boldsymbol{\beta}) \sum_{n=m}^{\infty} \frac{n!}{m!(n-m)!} d^m (1-d)^{n-m} \cdot \left(\gamma \frac{\lambda^n}{n!} e^{-\lambda} + (1-\gamma)\delta(n=0) \right) \\
&= P(y \mid m, \alpha, \boldsymbol{\beta}) \left(\gamma \frac{d^m \lambda^m}{m!} e^{-\lambda d} \sum_{n=m}^{\infty} \frac{\lambda^{n-m}(1-d)^{n-m}}{(n-m)!} e^{-\lambda(1-d)} + (1-\gamma)\delta(m=0) \right) \\
&= P(y \mid m, \alpha, \boldsymbol{\beta}) \left(\gamma \frac{(\lambda d)^m}{m!} e^{-\lambda d} + (1-\gamma)\delta(m=0) \right) \qquad (7.4) \\
&= P(y \mid m, \alpha, \boldsymbol{\beta}) \mathrm{PoiMix}(m; \lambda d, \gamma). \qquad (7.5)
\end{aligned}
$$

Equation (7.4) gives us the marginal distribution of a single value of m and y. To compute the posterior probability of the hyperparameters, we need the likelihood $P(\mathbf{m}, \mathbf{y} \mid \alpha, \boldsymbol{\beta}, \lambda, \gamma)$ of all the observed m and y values. Let $\mathcal{N} := \{i : m_i > 0\}$, $\mathcal{Z} := \{i : m_i = 0\}$, and $S = \sum_{i=1}^{r} m_i$. The joint probability of the m_i's can be somewhat simplified:

$$
\begin{aligned}
P(\mathbf{m} \mid \lambda, \gamma) &= \prod_i P(m_i \mid \lambda, \gamma) \\
&= \prod_i \left[\gamma \frac{(\lambda d)^{m_i}}{m_i!} e^{-\lambda d} + (1-\gamma)\delta(m_i = 0) \right] \\
&= \prod_{i \in \mathcal{N}} \gamma \frac{(\lambda d)^{m_i}}{m_i!} e^{-\lambda d} \cdot (1 - (1 - e^{-\lambda d})\gamma)^{|\mathcal{Z}|} \\
&= \gamma^{|\mathcal{N}|} e^{-\lambda d |\mathcal{N}|} \frac{(\lambda d)^S}{\prod_i m_i!} (1 - (1 - e^{-\lambda d})\gamma)^{|\mathcal{Z}|}. \qquad (7.6)
\end{aligned}
$$

The joint probability $P(\mathbf{y} \mid \mathbf{m}, \alpha, \boldsymbol{\beta})$ can also be simplified:

$$\begin{aligned} P(\mathbf{y} \mid \mathbf{m}, \alpha, \boldsymbol{\beta}) &= \prod_i [\beta_1 \text{Bin}(y_i; m_i, \alpha) + \beta_2 \delta(y_i = 0) + \beta_3 \delta(y_i = m_i)] \\ &= \prod_{i \in \mathcal{A}} \beta_1 \binom{m_i}{y_i} \alpha^{y_i} (1-\alpha)^{m_i - y_i} \\ &\quad \prod_{i \in \mathcal{B}} [\beta_1 (1-\alpha)^{m_i} + \beta_2] \\ &\quad \prod_{i \in \mathcal{C}} [\beta_1 \alpha^{m_i} + \beta_3], \end{aligned}$$

where

$$\begin{aligned} \mathcal{A} &:= \{i : m_i > y_i > 0\} \\ \mathcal{B} &:= \{i : m_i > y_i = 0\} \\ \mathcal{C} &:= \{i : m_i = y_i > 0\}. \end{aligned}$$

Note that the index sets $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are the same as the ones used in setting the hyperparameters c_1, c_2, and c_3.

7.3 Predicate Truth Posteriors

After setting the hyperparameters to their MAP estimates, we can proceed to our ultimate goal of computing the posterior truth probability $P(X > 0 \mid m, y)$.

The case where $y > 0$ is trivial since $P(X > 0 \mid m, Y > 0) = 1$. Hence we only need to examine the case where $Y = 0$. It is easier to first compute $P(X = 0 \mid m, Y = 0)$, from which we then obtain $P(X > 0 \mid m, Y = 0) = 1 - P(X = 0 \mid m, Y = 0)$.

We first compute the joint probability $P(X = 0, m, Y = 0)$, which is divided by $P(m, Y = 0)$ (Equation (7.4)) to get the desired answer. Keep in mind that the

hyperparameters have now been set to their MAP estimates.

$$
\begin{aligned}
P(X=0,m,Y=0) &= \sum_{n=m}^{\infty} P(n)P(m \mid n)P(X=0 \mid n)P(Y=0 \mid m,n,X=0) \\
&= \sum_{n=m}^{\infty} P(n)P(m \mid n)[\hat{\beta}_1 \text{Bin}(x=0;n,\hat{\alpha}) + \hat{\beta}_2\delta(x=0) + \hat{\beta}_3\delta(x=n)] \cdot 1 \\
&= \sum_{n=m}^{\infty} P(n)P(m \mid n)[\hat{\beta}_1(1-\hat{\alpha})^n + \hat{\beta}_2 + \hat{\beta}_3\delta(n=0)] \\
&= \hat{\beta}_1 P_1(X=0,m,Y=0) + \hat{\beta}_2 \text{PoiMix}(m;\hat{\lambda}d,\hat{\gamma}) + \hat{\beta}_3\delta(m=0)P(n=0),
\end{aligned}
$$

where

$$
\begin{aligned}
P_1(X=0,m,Y=0) &= \sum_{n=m}^{\infty} P(n)P(m \mid n)P(X=0 \mid n)P(Y=0 \mid X=0,m,n) \\
&= \sum_{n=m}^{\infty} P(n)P(m \mid n)\binom{n}{0}\hat{\alpha}^0(1-\hat{\alpha})^n \frac{\binom{0}{0}\binom{n}{m}}{\binom{n}{m}} \\
&= \sum_{n=m}^{\infty} P(n)P(m \mid n)(1-\hat{\alpha})^n \\
&= \sum_{n=m}^{\infty} \frac{n!}{m!(n-m)!} d^m(1-d)^{n-m}(1-\hat{\alpha})^n \left(\hat{\gamma}\frac{\hat{\lambda}^n}{n!}e^{-\hat{\lambda}} + (1-\hat{\gamma})\delta(n=0)\right) \\
&= \hat{\gamma}\frac{(\hat{\lambda}d(1-\hat{\alpha}))^m}{m!}e^{-\hat{\lambda}} \sum_{n=m}^{\infty} \frac{(\hat{\lambda}(1-d)(1-\hat{\alpha}))^{n-m}}{(n-m)!} + (1-\hat{\gamma})\delta(m=0) \\
&= \hat{\gamma}\frac{(\hat{\lambda}d(1-\hat{\alpha}))^m}{m!}e^{-\hat{\lambda}(1-(1-\hat{\alpha})(1-d))} + (1-\hat{\gamma})\delta(m=0). \qquad (7.7)
\end{aligned}
$$

We plug $Y=0$ into Equation (7.4) to obtain

$$
P(m,Y=0) \;=\; [\hat{\beta}_1(1-\hat{\alpha})^m + \hat{\beta}_2 + \hat{\beta}_3\delta(m=0)] \cdot [\hat{\gamma}\frac{(\hat{\lambda}d)^m}{m!}e^{-\hat{\lambda}d} + (1-\hat{\gamma})\delta(m=0)]. \qquad (7.8)
$$

Finally, dividing Equation (7.8) into Equation (7.7), we get

$$P(X=0 \mid m, Y=0) = \begin{cases} \frac{\hat{\beta}_1(1-\hat{\alpha})^m e^{-\hat{\lambda}\hat{\alpha}(1-d)}+\hat{\beta}_2}{\hat{\beta}_1(1-\hat{\alpha})^m+\hat{\beta}_2}, & \text{if } m>0, \\ \frac{(\hat{\beta}_1 e^{-\hat{\lambda}\hat{\alpha}(1-d)}+\hat{\beta}_2+\hat{\beta}_3 e^{-\hat{\lambda}(1-d)})\hat{\gamma}e^{-\hat{\lambda}d}+(1-\hat{\gamma})}{\hat{\gamma}e^{-\hat{\lambda}d}+(1-\hat{\gamma})}, & \text{if } m=0. \end{cases}$$

We can also compute $P(X>0 \mid n, m, Y=0)$, which is useful in validating our model.

$$\begin{aligned} P(n, m, Y=0, X=0) &= P(n)P(X=0 \mid n)P(m \mid n)P(Y=0 \mid X=0, m, n) \\ &= P(n)\text{Bin}(m; n)[\hat{\beta}_1(1-\hat{\alpha})^n + \hat{\beta}_2 + \hat{\beta}_3\delta(n=0)], \\ P(X=0 \mid n, m, Y=0) &= \frac{P(n, m, Y=0, X=0)}{P(n, m, Y=0)} \\ &= \frac{\hat{\beta}_1(1-\hat{\alpha})^n + \hat{\beta}_2 + \hat{\beta}_3\delta(n=0)}{\hat{\beta}_1\text{Bin}(y; m, \alpha) + \hat{\beta}_2\delta(y=0) + \hat{\beta}_3\delta(y=m)} \\ &= \frac{\hat{\beta}_1(1-\hat{\alpha})^n + \hat{\beta}_2 + \hat{\beta}_3\delta(n=0)}{\hat{\beta}_1(1-\hat{\alpha})^m + \hat{\beta}_2 + \hat{\beta}_3\delta(m=0)} \\ &= \begin{cases} 1, & \text{if } m=n=0 \\ \hat{\beta}_1(1-\hat{\alpha})^n + \hat{\beta}_2, & \text{if } m=0,\ n>0 \\ \frac{\hat{\beta}_1(1-\hat{\alpha})^n+\hat{\beta}_2}{\hat{\beta}_1(1-\hat{\alpha})^m+\hat{\beta}_2}, & \text{if } m>0,\ n>0. \end{cases} \end{aligned} \tag{7.9}$$

In deriving Equation (7.9), we made use of the formula for $P(n, m, y)$ from Equation (7.3).

Chapter 8

Multi-Bug Results

After inferring the predicate truth probabilities, we plug them into the predicate-run connection weight matrix A_{ij} and run the multi-bug algorithm. In this chapter, we present experimental results on the programs BC, CCRYPT, MOSS, RHYTHMBOX, and EXIF. Details of the datasets, including program size, deployed instrumentation schemes, as well as information regarding the bugs, may be found in section 3.5.

Table 8.1 summarizes the number of predicates selected in each stage of our multi-bug algorithm. The first column on the right hand side lists the initial number of predicates contained in each program. The second column lists the number of predicates left after pre-filtering. The third column lists the number of predicates that received a non-zero vote in the final voting stage of Algorithm 1. The last column lists the number of predicates needed to account for over 90% of the failed runs for each program.

In the typical usage scenario, the ranked predicate list would be examined manually by a test engineer who may quickly lose her patience the farther down the list she goes. Thus a good bug analysis algorithm should return a predicate list that is as con-

Table 8.1: Summary statistics for multi-bug algorithm results.

	Predicate Counts			
	Initial	Pre-Filtered	Voted	90% Failures
CCRYPT	58,720	50	1	1
BC	298,482	147	14	1
MOSS	202,998	2645	62	14
RHYTHMBOX	857,384	537	82	6
EXIF	156,476	272	7	2

cise as possible, with the most important bugs listed at the top. In our experiments, Algorithm 1 filters out over 99.9% of predicates for all of the tested programs. In the results for RHYTHMBOX and MOSS, however, there are still tens of predicates that received a vote in the final stage of the multi-bug algorithm. However, as the last column of the table indicates, over 90% of failed runs in all five test programs were accounted for by the top few predicates. Hence a test engineer who uses our system probably would not have to examine too many predicates to find the bulk of failure-inducing bugs.

Let us examine the results in detail. For each dataset, we list as many of the top predicates as is necessary to cover 90% of the failed runs, ranked by the number of failed runs it accounts for. For each test program, the first column of the results table lists the number of failed runs attributed to that predicate. The rest of the columns contain additional information about the predicates.

8.1 CCRYPT

There are 10316 failed runs in the CCRYPT dataset, all of which are accounted for by a single predicate. The predicate `line <= outfile___0` in Table 8.2 is equivalent to the one selected by the single-bug algorithm. In the `prompt()` function, the

Table 8.2: CCRYPT failure predictors given by the multi-bug algorithm.

# Failures	Predicate	Function
10,316	`line <= outfile___0`	`prompt()`

Table 8.3: BC failure predictors given by the multi-bug algorithm.

# Failures	Predicate	Function
7153	`a_names < v_names`	`more_arrays()`

return value to `xreadline()` is stored in the pointer variable `line`. The variable `outfile___0` represents the value zero. The selected CCRYPT predicate indicates that `line` being less than or equal to zero can account for all of the failed runs. Indeed, the CCRYPT bug occurs because the function `xreadline()` returns `null` when the user inputs `EOF`. This `null` return value is unchecked; the program subsequently fails when it tries to access the string that `xreadline()` supposedly returned.

8.2 BC

Table 8.3 indicates that 7135 out of 7802 of the BC failed runs are accounted for by the predicate `a_names < v_names`. This predicate signifies an anomaly that only occurs along with the array overrun. As a bug predictor, it is not as direct as, say, the predicate `index > a_count`. But it is in fact a more deterministic predictor for this non-deterministic bug, and one that accounts for more failed runs.

8.3 MOSS

Table 8.4 contains the failure predictors for MOSS. The left hand side lists the ranked predicates along with the number of failed runs they account for. The right hand side

is a bug histogram showing the bug counts of the failed runs where the predicates are true. Note that this bug histogram tallies all the runs in which the predicate is true, not just those attributed to this predicate. Later on we will see that there is an important difference between the two sets of bug histograms.

The first predicate indicates that the input file language is LISP. This is a direct predictor for bug #5. We can verify from the bug histogram that all 1572 runs attributed to this predicate crashed due to bug #5. There are few other bugs with non-zero counts in this histogram, but that is because a single run may trigger multiple bugs. What is important is that all the crash due to bug #5 are correctly attributed to the right predicate.

As mentioned in subsection 3.5.3, bug #5 is the easiest bug to isolate in MOSS because it lies on a completely different code path from the rest of the bugs. Hence our success at selecting the right predicate for bug #5, while comforting, is not altogether surprising.

Continuing down the list, the second predicate is a direct predictor for bug #4, and it is shown to account for most of the runs containing the bug. The relevant code fragment is reproduced below:

```
config.tile_size = atoi(argv[++i]);
config.token_window_size =  config.tile_size+1;
```

The variable `config.tile_size` is an integer representing the size of tiles in MOSS. The size of the token window is set to be one larger than the size of the tile. Bug #4 is an array overrun bug that could occur if `config.token_window_size` is larger than the maximum allowed token window size, which is set to 500. Thus the second predicate, `config.tile_size > 500` signifies a direct violation of the size constraint; it is a direct predictor of the bug.

The third predicate in the list accounts for a subset of bug #1, the C comment-

matching bug. Strictly speaking, predicate 3 a sub-bug predictor for bug #1, because it does not account for all runs with the bug. Predicate 7 further down the list is another sub-bug predictor for bug #1. The two predicates are located on different lines in the source code. Together, they account for most of the bug #1 runs. Bug #1 occurs in C-style comment-matching mode: newlines within comments are erroneously ignored, resulting in incorrect line numbers in the output file. Even though they are sub-bug predictors, the selected predicates are clearly indicate the location of the bug and are therefore acceptable predictors for the bug.

Predicates 4 and 6 are both sub-bug predictors of bug #9. Bug #9 is an array overrun bug that is very similar to bug #4: the size of a window is set to be larger than the allow maximum of 100. Predicate 4 (`__lengthofp == 200`) indicates that the length of the window `p` is too long, whereas predicate 6 (`i___0 > 500`) is true when a the window index `i` is large. Both predicates clearly point out the problem with this window array, and are thus useful predictors for bug #9.

Predicate 5 (`i > 52`) accounts for bug #3. This bug occurs when more than one database is open. Each database contains multiple files. The predicate indicates that the number of processed files is larger than usual, a fact that is correlated with multiple databases being open.

Predicate 8 (`f < yyout`) accounts for bug #2, which is a bug similar to the one in CCRYPT. The program opens a file for writing, but neglects to check the outcome of the function `fopen()` to ensure that the file is writable. When there is an error, the variable `f` is zero, which is less than the value of `yyout`. Predicate 8 is a direct predictor for bug #2.

We now come to predicate 9. A glance at the bug histogram should raise immediate concern. The bug histogram indicates that this is a super-bug predictor. The predicate is true when the command-line contains more than 8 elements, a condition shared by a large number of failed runs as well as successful runs.

Table 8.4: MOSS failure predictors with overall bug histogram.

		Bug Histogram of Runs where Predicate is True							
# Failures	Predicate	#1	#2	#3	#4	#5	#6	#7	#9
1572	`files[filesindex].language > 16`	0	0	25	57	1572	0	0	70
880	`config.tile_size > 500`	26	0	12	940	57	0	0	51
626	`config.match_comment is TRUE`	796	6	9	6	0	7	3	45
489	`__lengthofp == 200`	40	5	13	6	10	5	8	577
391	`i > 52`	13	0	436	0	0	0	2	23
301	`i___0 > 500`	25	4	10	0	0	0	1	350
222	`config.match_comment is TRUE`	1155	11	14	2	0	7	6	65
146	`f < yyout`	11	146	3	0	0	1	0	10
134	`i >= 8`	1194	146	497	120	211	259	9	212
70	`((*(fi + i)))->this.last_line < 4`	761	0	9	0	0	0	7	9
58	`__lengthofp > 500`	26	5	10	7	8	0	1	430
34	`((*(fi + i)))->other.last_line == yy_start`	710	0	8	0	0	0	5	3
28	`((*(fi + i)))->this.last_line == 1`	720	0	7	0	0	0	5	7
28	`((*(fi + i)))->this.last_line < 2`	721	0	7	0	0	0	6	7

To understand why this predicate was chosen by the algorithm, let us take a look at a different set of bug histograms. In Table 8.5, we tally only the runs attributed to each predicate. Notice that this set of bug histograms is a lot cleaner than the previous one. This indicates that, while the selected predicates may be true in runs that fail due to various bugs, the runs attributed to each predicate contain roughly the same bugs. Exceptions are bugs 1 and 9, each of which are fragmented across a few distinct predicates. But overall, these bug histograms reveal fairly tight clusters for the failed runs.

Let us examine the bug histogram for predicate 9 (`i > 8`). Even though this predicate is a super-bug predictor, here it only accounts for runs crashing due to bug #6. This is somewhat comforting. But why didn't the algorithm pick a better predictor for bug #6? The answer lies in our definition of a good bug predictor. Bug #6 occurs when the "`-p`" option is passed to the command-line. Thus a good predictor for bug # 6 is a branch statement in function `handle_options()` testing for the "`-p`" option. However, even if this condition is `FALSE`, there is still a good chance that the program would fail, because other command-line options may still trigger other bugs. This violates our assumption that predicates P and $\bar{P}$ cannot

Table 8.5: MOSS failure predictors with bug histogram of accounted runs.

		Bug Histogram of Runs Attributed to Predicate							
# Failures	Predicate	#1	#2	#3	#4	#5	#6	#7	#9
1572	`files[filesindex].language > 16`	0	0	25	57	1572	0	0	70
880	`config.tile_size > 500`	26	0	11	880	0	0	0	46
626	`config.match_comment is TRUE`	626	0	0	0	0	0	3	44
489	`__lengthofp == 200`	0	0	0	0	0	4	8	489
391	`i > 52`	0	0	391	0	0	0	2	18
301	`i___0 > 500`	0	0	0	0	0	0	1	301
222	`config.match_comment is TRUE`	222	0	0	0	0	2	3	20
146	`f < yyout`	11	146	3	0	0	1	0	10
134	`i >= 8`	0	0	0	0	0	134	1	0
70	`((*(fi + i)))->this.last_line < 4`	70	0	0	0	0	0	0	0
58	`__lengthofp > 500`	0	0	0	0	0	0	0	58
34	`((*(fi + i)))->other.last_line == yy_start`	34	0	0	0	0	0	0	0
28	`((*(fi + i)))->this.last_line == 1`	28	0	1	0	0	0	0	0
28	`((*(fi + i)))->this.last_line < 2`	28	0	0	0	0	0	0	0

both be good bug predictors.

This also explains why none of the predictors in `handle_options()` was selected by the algorithm. For example, bug #1 could have been predicted by the "`-c`" option, but is now predicted instead by the two afore-mentioned sub-bug predictors. In the case of bug #6, the strength of the "`-p`" predicate is offset by the strength of its complement. In comparison, the super-bug `i > 8` appeared stronger, and was chosen by the algorithm.

This caveat underscores the fact that there is no safe assumption one can make about all possible types of bugs. In many cases, our definition of a good bug predictor works well. But there are situations where it is less appropriate.

8.4 Obtaining Predicate Clusters from Run Clusters

Our multi-bug algorithm picked a non-ideal predictor for bug #6 in MOSS. But all is not lost. Notice that the algorithm nevertheless yields very good run clusters. Thus the constituency of runs for each selected predicate can be treated as a single bug cluster. The availability of good run clusters allows us to reduce the multi-bug

Table 8.6: Predicate rankings based on predicate 9 of original MOSS results with overall bug histogram.

		Bug Histogram of Runs where Predicate is True							
Rank	Predicate	#1	#2	#3	#4	#5	#6	#7	#9
1	`tmp___8 == 0 is TRUE`	11	1	0	11	4	259	1	15
2	`strcmp == 0`	11	1	0	11	4	259	1	15
3	`i == 9`	0	129	0	65	106	227	1	98
4	`i == 7`	0	129	0	65	106	227	1	98
...									

problem into single-bug cases. We can compare each cluster separately against the set of successful runs, ranking the predicates using, say, a simple univariate test. Doing so would also demonstrate predicate correlation patterns. Thus we can obtain predicate clusters from the run clusters.

We use the same two-sample T-test from our earlier attempt in section 5.1, this time with the aid of single-bug run clusters. Let $\mathcal{F}_i$ denote the set of failed runs attributed to predicate i, as determined by the multi-bug algorithm. Let $\mathcal{S}$ denote the set of successful runs. For each top predicate i, we rank all the predicates based on the two-sample T statistic

$$T(\mathcal{F}_i, \mathcal{S}) = \frac{\hat{\pi}_{\mathcal{F}_i} - \hat{\pi}_{\mathcal{S}}}{\mathrm{Var}_{\mathcal{F}_i,\mathcal{S}}},$$

where $\hat{\pi}_{\mathcal{F}_i} = \sum_{j\in\mathcal{F}_i} P(X_{ij} > 0 \mid m_{ij}, y_{ij})/|\mathcal{F}_i|$ and $\hat{\pi}_{\mathcal{S}} = \sum_{j\in\mathcal{S}} P(X_{ij} > 0 \mid m_{ij}, y_{ij})/|\mathcal{S}|$ are the average inferred truth probabilities. We rank the predicates in the cluster by the value of their T statistic.

Table 8.6 contains the predicate ranking based on the run cluster of predicate 9, the super-bug predictor for bug #6 in Table 8.4. The first two predicates are direct predictors for bug #6 that indicate that the commandline flag "`-p`" is set. The rest are super-bug predictors having to do with the length of the command-line.

We can also take a look at some of the other predicate clusters. Table 8.7 gives us

Table 8.7: Predicate rankings based on predicate 3 of original MOSS results with overall bug histogram.

		Bug Histogram of Runs where Predicate is True							
Rank	Predicate	#1	#2	#3	#4	#5	#6	#7	#9
1	`config.match_comment is TRUE`	796	6	9	6	0	7	3	45
2	`i == 3`	1194	13	18	43	79	12	6	87
3	`i == 5`	1194	13	18	43	79	12	6	87
4	`i == 7`	1194	13	18	43	79	12	6	87
5	`i == 9`	1194	13	18	43	79	12	6	87
6	`tmp___0 == 0 is TRUE`	1194	13	18	43	79	12	6	87
7	`strcmp == 0`	1194	13	18	43	79	12	6	87
8	`i == lineno`	1183	13	18	42	79	0	6	87
9	`i == yy_init`	1183	13	18	42	79	0	6	87
10	`i == 1`	1183	13	18	42	79	0	6	87
11	`i < 2`	1183	13	18	42	79	0	6	87
12	`config.match_comment is TRUE`	1155	11	14	2	0	7	6	65
...									

the cluster for `config.match_comment is TRUE`, the sub-bug predictor for bug #1. The bug histogram shows that the top ranked predicates in this table are all sub-bug predictors for bug #1. In particular, predicate 7 in the original list is ranked 12 in this cluster, showing that these two top-ranked predicates can indeed be grouped together. The true predictors for bug #1 is also contained in the list: predicates 6 and 7 in this list have to do with the command-line option "`-c`", which puts the program in C comment-matching mode. Incidentally, these bug histograms also show that these predictors are equivalent to the `i == ...` command-line length predicates. This is an artifact of our data-generation process.

We also look at the cluster for predicate 4, the sub-bug predictor of bug #9. Table 8.8 contains many other high-quality sub-bug predictors of bug #9, along with the actual bug predictor, `config.winnowing_window_size >= 200`. We see that the sub-bug predictor is in fact a more deterministic predictor for bug #9; it is true in 1011 bug #9 runs – many more runs than `config.winnowing_window_size >= 200`.

Table 8.8: Predicate rankings based on predicate 4 of original MOSS results with overall bug histogram.

		Bug Histogram of Runs where Predicate is True							
Rank	Predicate	#1	#2	#3	#4	#5	#6	#7	#9
1	`__lengthofp == 200`	40	5	13	6	10	5	8	577
2	`config.winnowing_window_size == 200`	41	5	15	29	42	5	8	631
3	`__lengthofp > 50`	66	10	25	16	19	5	9	1011
4	`__lengthofp >= 200`	65	10	23	12	17	5	9	1004
5	`config.winnowing_window_size >= 200`	67	10	26	51	70	5	9	1095
6	`passage_index___0 > 128`	67	10	23	7	0	5	9	986
...									

8.5 RHYTHMBOX

There are at least two bugs in RHYTHMBOX, one of which exposes a bad coding pattern that has to do with previously freed and reclaimed event objects. This bug subsequently led to the discovery of multiple bugs within RHYTHMBOX resulting from the same bad coding practice. This in part explains the high failure rate in our RHYTHMBOX dataset. (Table 3.2 shows that over 60% of our RHYTHMBOX runs are crashes.)

In Table 8.9, the first three predicates account for over 85% of the failed runs. The first predicate is an important clue for the disclosure animation bug. The bug involves a dangling pointer to an object that has already been destroyed. The predicate indicates that there are inconsistencies in the timer object.

The second predicate is a predictor of the race condition bug that has to do with initialization of the rhythmbox player. Table 8.10 shows the cluster for predicate 2. We see that `monkey_media_player_get_uri() == NULL`, a smoking gun predictor, is clearly grouped together with this predicate.

The rest of the predicates on the list in Table 8.9 are all manifestations of other instances of the bad coding practice we had revealed.

Table 8.9: RHYTHMBOX failure predictors given by the multi-bug algorithm.

# Failures	Predicate	Function
11,658	`(mp->priv)->timer is FALSE`	`monkey_media_player_finalize()`
2566	`tmp___5 is FALSE`	`info_available_cb()`
2319	`vol <= (float) 0 is TRUE`	`rb_volume_sync_volume()`
623	`(db->priv)->thread_reaper_id >= 12`	`rhythmdb_init()`
316	`rorder (new val) < rorder (old val)`	`rb_random_play_order_by_age()`
303	`(mp->priv)->tick_timeout_id > 12`	`monkey_media_player_init()`

Table 8.10: Predicate rankings based on predicate 2 of original RHYTHMBOX results.

Rank	Predicate	Function
1	`tmp___5 is FALSE`	`info_available_cb()`
2	`monkey_media_player_get_uri() == 0`	`info_available_cb()`
2	`monkey_media_player_get_uri() == 0`	`info_available_cb()`
...		

8.6 EXIF

The results for EXIF are shown in Table 8.11. Here, too, we can make use of the predicate clusters to analyze each of the high-ranked predicates. Judging by their predicate clusters (Table 8.12 and Table 8.14), predicates 1 and 3 both predict the `NULL`-printing bug. In the machine readable mode, when the variable `maxlen` is greater than a certain number in the `exif_entry_get_value()` function, the function returns `NULL`, which eventually gets passed to a `printf()` function and crashes. The predicate `maxlen > 1900` is number 4 on the cluster list for predicate 1, and number 1 on the list for predicate 3. Another good predictor of this bug is ranked number 3 in Table 8.14; it indicates that `exif_entry_get_value()` is returning `NULL`. The rest of the predicates in these two clusters are equivalent (but redundant) indicators of the same problem.

Predicate 2 from the original multi-bug algorithm results is a direct indicator of the `memmove()` bug. The program crashes when `memmove()` receives a negative byte count, which is represented by the predicate `i < 0`.

Table 8.11: EXIF failure predictors given by the multi-bug algorithm.

# Failures	Predicate	Function
1644	`i < k`	`exif_entry_get_value()`
532	`i < 0`	`jpeg_data_set_exif_data()`
17	`machine_readable is TRUE`	`main()`
12	`(data->ifd[4])->count is FALSE`	`exif_data_save_data_content()`

Table 8.12: Predicate rankings based on predicate 1 of original EXIF results.

Rank	Predicate	Function
1	`i < k`	`exif_entry_get_value()`
2	`i < k`	`exif_entry_get_value()`
3	`i <= k`	`exif_entry_get_value()`
4	`maxlen > 1900`	`exif_entry_get_value()`
5	`i < k`	`exif_entry_get_value()`
...		

The fourth predicate in the original list, `(data->ifd[4])->count is FALSE`, is a secondary indicator of the last bug in EXIF, which has to do with loading images taken by Canon cameras. The primary indicator of the bug, `o + s > buf_size`, is ranked number 1 on the cluster list in Table 8.15. There are very few crashes due to this bug. Hence its predictor is ranked behind the predictors for the other two bugs.

Table 8.13: Predicate rankings based on predicate 2 of original EXIF results.

Rank	Predicate	Function
1	`i < 0`	`jpeg_data_set_exif_data()`
2	`i < 0 is TRUE`	`jpeg_data_set_exif_data()`
3	`i < 52`	`jpeg_data_set_exif_data()`
4	`i < 60`	`jpeg_data_set_exif_data()`
...		

Table 8.14: Predicate rankings based on predicate 3 of original EXIF results.

Rank	Predicate	Function
1	`maxlen > 1900`	`exif_entry_get_value()`
2	`maxlen >= 72`	`exif_entry_get_value()`
3	`tmp___873 is FALSE`	`exif_entry_get_value()`
...		

Table 8.15: Predicate rankings based on predicate 4 of original EXIF results.

Rank	Predicate	Function
1	`o + s > buf_size is TRUE`	`exif_mnote_data_canon_load()`
2	`(data->ifd[4])->count is FALSE`	`exif_data_save_data_content()`
...		

8.7 Summary of Results

Our results strongly indicate the usefulness of a statistical approach to bug-finding. In many cases, our algorithm returns direct indicators of the bug. In some cases, the algorithm prefers more deterministic predictors rather than the smoking-gun causes of the bugs. While these deterministic predictors are not as direct at indicating the problem, they do reveal interesting information about the failure modes of the bugs, and are therefore useful for the bug-finding process.

We learn two important lessons from our results. First, we learn that there are perhaps no universally-valid assumptions about characteristics of bug predictor characteristics. Even reasonable sounding assumptions—such as P and $\bar{P}$ not both being good bug predictors—are inappropriate on occasion.

Secondly, we learn that having a run cluster and a predicate cluster are very helpful during debugging. In some sense, run clustering is the necessary evil and a precondition for obtaining sensible predicates. Predicate clusters group together all the redundant predictors of a bug. It would be nice if our algorithm can select

the "best" predicate out of each cluster. But often times, statistics alone cannot determine the merit of a predicate. Such a decision requires semantic analysis of the program source code.

However, it is always useful to present correlation lists to users of the system. After all, the programmer who wrote the code is the best candidate for distinguishing amongst redundant predictors. It is often unsatisfactory for a user of the system to be bluntly confronted by a list of suspicious predicates. Such a list may be concise, but often does not contain enough information to fully describe the failure modes. Having the predicate correlation clusters allows the user to examine the failure modes in more detail, thereby allowing for more efficient debugging. When using our own system to debug programs with unknown bugs, we ourselves have found it exceedingly useful to have such correlation information between predicates.

Our multi-bug algorithm is a three-stage process: pre-filtering, iterative predicate voting, and post-processing. Pre-filtering reduces computational costs, and post-processing gives us useful predicate clusters. Both involve some form of univariate hypothesis testing. In section 5.1, we demonstrated how simple univariate hypothesis testing by itself does not provide a good predicate ranking. The strength of a good predictor for one bug is often diluted by the presence of other bugs. High-ranked predicates from the two-sample T test are super-bug predictors. Thus pre-filtering or post-processing alone cannot select high-quality predicates. In order to isolate predictors for multiple bugs simultaneously, we must first have either good run clusters or predicate clusters. Our multi-bug algorithm gives us the former.

It is also worth noting that having a good predicate truth-probability model is very helpful for the analysis algorithm. The truth probabilities are used as connection weights between predicates and runs. In some cases the inferred truth probabilities do form spurious connections where one does not exist. But overall, using inferred truth probabilities gives us higher quality results than using the observed predicate

Table 8.16: Comparison of average absolute errors of inferred truth probabilities vs. observed truth counts.

	CCRYPT	BC	MOSS	RHYTHMBOX	EXIF
Inferred	3.26e-18	0.1398	0.1767	0.02424	0.1258
Observed	0	0.1447	0.1809	0.02571	0.1284

samples. Table 8.16 contains the average absolute errors between the inferred truth probabilities and the real (unsampled) truth count, compared against that of the observed truth counts and the real truth count. The inferred truth probabilities have lower average absolute errors in four out of five programs. The exception occurs in CCRYPT, where the difference between the two error rates is negligible.

From a debugging perspective, the predicate lists presented here are of comparable quality to earlier results presented in Liblit *et al.* [2005]. However, from an algorithmic perspective, Algorithm 1 is built on a more formal framework, and would therefore be more conducive to theoretical studies of the statistical debugging approach. For instance, it would be interesting to see whether the algorithm is maximizing some underlying cost function. It would also be useful to determine the "power" of our algorithm, and to come up with rough figures for how many runs might be needed to catch certain percentages of certain types of bugs. We have done some preliminary experiments along this line of research, but a thorough theoretical investigation would be much more elucidating.

Chapter 9

Conclusions

In this thesis, we introduced the statistical debugging problem, defined the data collection framework, discussed the idiosyncrasies of bug-finding and presented two algorithms that work well on real-world programs. Our journey on the road of automatic software debugging is far from over. There are many possiblities for future work. We have gathered experience by working with a few programs. While they contain bugs that are representative of common software bugs one might encounter, they do not cover the whole spectrum. There are many other kinds of bugs that are not addressed by our current methods. We intend to both broaden our scope of experimentation, as well as improving upon the results we already have.

Part of our focus lies in bugs that are traditionally difficult to catch at run time, such as memory corruption bugs. While there exist static program analysis tools for catching memory leakage and corruption bugs, they are often quite expensive to run. However, they do present an alternative. Hence catching memory corruption bugs alone does not justify our debugging approach.

There are many other situations where a combination of dynamic analysis and statistical analysis may be the only option. We touch upon this territory with the event

timing-related bugs in RHYTHMBOX. In general, race conditions and deadlocks in multi-threaded programs are notoriously difficult to debug. Our predicate instrumentation and sampling framework is not particular suited for observing such rare events, especially when they are also timing-dependent. This may call for modifications to our instrumentation schemes and data recording method.

New types of bugs and instrumentation schemes present new challenges to the debugging algorithm. This project has been an adventure filled with interaction between programming analysis needs and machine learning mights. In designing the analysis algorithms, we paid considerable attention to the morphology of commonly encountered bugs, and sought inspiration from programming analysis heuristics.

We believe that more can be done to incorporate classic programming analysis techniques in the statistical analysis of programs. We began investigating this approach by bringing chronological ordering of predicates into the analysis algorithm. Preliminary results are not ideal. But the combination of statistical and semantic program analysis remains the only promising candidate for selecting the best bug predictor from redundant predicates.

As computer systems become increasingly complex, more and more effort will surely go into automatic management and troubleshooting of such systems. In the coming years, we expect to see a lot more similar collaborations between statistical machine learning and traditional computer science. Our findings in this project highlight the potential benefits of such a collaboration to both fields. This, we hope, is just the beginning.

Appendices

Appendix A

Parameter Estimates in the Predicate Truth-Value Model

We derive the necessary formulas for obtaining the MAP estimate of the prior parameters in the predicate truth probability model in chapter 7. This requires calculating the the gradient and the Hessian of the Bayesian likelihood function Equation (7.1). As explained in section 7.2, the posterior probability of the prior parameters given observations $\mathbf{m}$ and $\mathbf{y}$ is factorable, which simplifies the calculation of the cross-terms in the Hessian.

Starting from Equation (7.1), we have

$$
\begin{aligned}
(\hat{\alpha}, \hat{\boldsymbol{\beta}}, \hat{\gamma}, \hat{\lambda}) &= \operatorname{argmax}_{\alpha,\boldsymbol{\beta},\gamma,\lambda} \log P(\alpha, \boldsymbol{\beta}, \gamma, \lambda, \mathbf{m}, \mathbf{y} \mid d) && \text{(A.1)} \\
&= \operatorname{argmax}_{\alpha,\boldsymbol{\beta},\gamma,\lambda} \log P + \log Q && \text{(A.2)} \\
&= (\operatorname{argmax}_{\alpha,\boldsymbol{\beta}} \log P, \operatorname{argmax}_{\gamma,\lambda} \log Q), && \text{(A.3)}
\end{aligned}
$$

where

$$\begin{aligned}
\log P(\mathbf{y}, \alpha, \boldsymbol{\beta} \mid \mathbf{m}) &= \log \frac{\Gamma(s+t)}{\Gamma(s)\Gamma(t)} + \log \frac{\Gamma(\sum_i c_i)}{\prod_i \Gamma(c_i)} && \text{(A.4)}\\
&\quad +(s-1)\log(1-\alpha) + (t-1)\log\alpha && \text{(A.5)}\\
&\quad +(c_1-1)\log\beta_1 + (c_2-1)\log\beta_2 + (c_3-1)\log\beta_3 && \text{(A.6)}\\
&\quad + \sum_{i\in\mathcal{A}} \left[\log\beta_1 + \log\binom{m_i}{y_i} + y_i\log\alpha + (m_i - y_i)\log(1-\alpha)\right] && \text{(A.7)}\\
&\quad + \sum_{i\in\mathcal{B}} \log[\beta_1(1-\alpha)^{m_i} + \beta_2] && \text{(A.8)}\\
&\quad + \sum_{i\in\mathcal{C}} \log[\beta_1\alpha^{m_i} + \beta_3], && \text{(A.9)}
\end{aligned}$$

and

$$\begin{aligned}
\log Q &= \log \frac{\lambda^{u-1}e^{-\lambda/v}}{\Gamma(u)v^u} + \log\frac{\Gamma(j+k)}{\Gamma(j)\Gamma(k)}(1-\gamma)^{j-1}\gamma^{k-1} && \text{(A.10)}\\
&\quad + \log\gamma^{|\mathcal{N}|}e^{-\lambda d|\mathcal{N}|}\frac{(\lambda d)^S}{\prod_i m_i!}(1-(1-e^{-\lambda d})\gamma)^{|\mathcal{Z}|} && \text{(A.11)}\\
&= (u-1)\log\lambda - \frac{\lambda}{v} + (j-1)\log(1-\gamma) + (k-1)\log\gamma && \text{(A.12)}\\
&\quad +|\mathcal{N}|\log\gamma - \lambda d|\mathcal{N}| + S\log\lambda + |\mathcal{Z}|\log(1-(1-e^{-\lambda d})\gamma) && \text{(A.13)}\\
&\quad +const. && \text{(A.14)}
\end{aligned}$$

Let's first optimize $\log P$ for $\hat{\alpha}$ and $\hat{\boldsymbol{\beta}}$. Let $B_i := \beta_1(1-\alpha)^{m_i} + \beta_2$ and $C_i := \beta_1\alpha^{m_i} + \beta_3$. In addition, we use the softmax representation for the β's:

$$\begin{aligned}
\beta_i &= \frac{e^{t_i}}{T}, && \text{(A.15)}\\
\text{where} \qquad T &= \sum_j e^{t_j}. && \text{(A.16)}
\end{aligned}$$

We first calculate the partial derivatives of β_i, B_i, and C_i:

$$d_{ij} := \frac{\partial \beta_i}{\partial t_j} = \frac{T e^{t_i} \delta(i=j) - e^{t_i} e^{t_j}}{T^2}, \tag{A.17}$$

$$D_{ijk} := \frac{\partial^2 \beta_i}{\partial t_j \partial t_k} \tag{A.18}$$

$$= \frac{T^2 e^{t_i} \delta(i=j=k) - T e^{t_i}(e^{t_j}\delta(i=k) + e^{t_j}\delta(j=k) + e^{t_k}\delta(i=j)) + 2e^{t_i}e^{t_j}e^{t_k}}{T^3}, \tag{A.19}$$

$$\frac{\partial B_i}{\partial t_j} = d_{1j}(1-\alpha)^{m_i} + d_{2j}, \tag{A.20}$$

$$\frac{\partial B_i}{\partial \alpha} = -\beta_1 m_i (1-\alpha)^{m_i - 1}, \tag{A.21}$$

$$\frac{\partial^2 B_i}{\partial t_j \partial t_k} = D_{1jk}(1-\alpha)^{m_i} + D_{2jk}, \tag{A.22}$$

$$\frac{\partial^2 B_i}{\partial t_j \partial \alpha} = -d_{1j} m_i (1-\alpha)^{m_i - 1}, \tag{A.23}$$

$$\frac{\partial^2 B_i}{\partial \alpha^2} = \beta_1 m_i (m_i - 1)(1-\alpha)^{m_i - 2}, \tag{A.24}$$

$$\frac{\partial C_i}{\partial t_j} = d_{1j}\alpha^{m_i} + d_{3j}, \tag{A.25}$$

$$\frac{\partial C_i}{\partial \alpha} = \beta_1 m_i \alpha^{m_i - 1}, \tag{A.26}$$

$$\frac{\partial^2 C_i}{\partial t_j \partial t_k} = D_{1jk}\alpha^{m_i} + D_{3jk}, \tag{A.27}$$

$$\frac{\partial^2 C_i}{\partial t_j \partial \alpha} = d_{1j} m_i \alpha^{m_i - 1}, \tag{A.28}$$

$$\frac{\partial^2 C_i}{\partial \alpha^2} = \beta_1 m_i (m_i - 1)\alpha^{m_i - 2}. \tag{A.29}$$

Thus, the first-order derivatives are:

$$\frac{\partial \log P}{\partial \alpha} = -\frac{s-1}{1-\alpha} + \frac{t-1}{\alpha} + \sum_{i \in \mathcal{A}} \left[\frac{y_i}{\alpha} - \frac{m_i - y_i}{1-\alpha} \right] + \sum_{i \in \mathcal{B}} B_i^{-1} \frac{\partial B_i}{\partial \alpha} + \sum_{i \in \mathcal{C}} C_i^{-1} \frac{\partial C_i}{\partial \alpha}, \tag{A.30}$$

$$\frac{\partial \log P}{\partial t_j} = \sum_{i=1,2,3} \frac{c_i - 1}{\beta_i} d_{ij} + \sum_{i \in \mathcal{A}} \beta_1^{-1} d_{1j} + \sum_{i \in \mathcal{B}} B_i^{-1} \frac{\partial B_i}{\partial t_j} + \sum_{i \in \mathcal{C}} C_i^{-1} \frac{\partial C_i}{\partial t_j}. \tag{A.31}$$

The second-order derivatives are:

$$\begin{aligned} \frac{\partial^2 \log P}{\partial \alpha^2} &= -\frac{s-1}{(1-\alpha)^2} - \frac{t-1}{\alpha^2} + \sum_{i \in \mathcal{A}} \left[-\frac{y_i}{\alpha^2} - \frac{m_i - y_i}{(1-\alpha)^2} \right] \\ &\quad + \sum_{i \in \mathcal{B}} -B_i^{-2} \left(\frac{\partial B_i}{\partial \alpha} \right)^2 + B_i^{-1} \frac{\partial^2 B_i}{\partial \alpha^2} \\ &\quad + \sum_{i \in \mathcal{C}} -C_i^{-2} \left(\frac{\partial C_i}{\partial \alpha} \right)^2 + C_i^{-1} \frac{\partial^2 C_i}{\partial \alpha^2}, \end{aligned} \tag{A.32}$$

$$\begin{aligned} \frac{\partial^2 \log P}{\partial t_j \partial t_k} &= \sum_{i=1,2,3} -\frac{c_i - 1}{\beta_i^2} d_{ik} d_{ij} + \frac{c_i - 1}{\beta_i} D_{ijk} \\ &\quad + \sum_{i \in \mathcal{A}} -\beta_1^{-2} d_{1k} d_{1j} + \beta_1^{-1} D_{1jk} \\ &\quad + \sum_{i \in \mathcal{B}} -B_i^{-2} \frac{\partial B_i}{\partial t_k} \frac{\partial B_i}{\partial t_j} + B_i^{-1} \frac{\partial^2 B_i}{\partial t_k \partial t_j} \\ &\quad + \sum_{i \in \mathcal{C}} -C_i^{-2} \frac{\partial C_i}{\partial t_k} \frac{\partial C_i}{\partial t_j} + C_i^{-1} \frac{\partial^2 C_i}{\partial t_k \partial t_j}, \end{aligned} \tag{A.33}$$

$$\begin{aligned} \frac{\partial^2 \log P}{\partial t_j \partial \alpha} &= \sum_{i \in \mathcal{B}} -B_i^{-2} \frac{\partial B_i}{\partial t_j} \frac{\partial B_i}{\partial \alpha} + B_i^{-1} \frac{\partial^2 B_i}{\partial t_j \partial \alpha} \\ &\quad + \sum_{i \in \mathcal{C}} -C_i^{-2} \frac{\partial C_i}{\partial t_j} \frac{\partial C_i}{\partial \alpha} + C_i^{-1} \frac{\partial^2 C_i}{\partial t_j \partial \alpha}. \end{aligned} \tag{A.34}$$

Lastly, we compute the derivatives of $\log Q$ for estimating $\hat{\lambda}$ and $\hat{\gamma}$:

$$\frac{\partial \log Q}{\partial \lambda} = \frac{u-1}{\lambda} - \frac{1}{v} + \frac{\partial L}{\partial \lambda}, \tag{A.35}$$

$$\frac{\partial \log Q}{\partial \gamma} = -\frac{j-1}{1-\gamma} + \frac{k-1}{\gamma} + \frac{\partial L}{\partial \gamma}, \tag{A.36}$$

$$\frac{\partial^2 \log Q}{\partial \lambda^2} = -\frac{u-1}{\lambda^2} + \frac{\partial^2 L}{\partial \lambda^2}, \tag{A.37}$$

$$\frac{\partial^2 \log Q}{\partial \gamma^2} = -\frac{j-1}{(1-\gamma)^2} - \frac{k-1}{\gamma^2} + \frac{\partial^2 L}{\partial \gamma^2}, \tag{A.38}$$

$$\frac{\partial^2 \log Q}{\partial \lambda \partial \gamma} = \frac{\partial^2 L}{\partial \lambda \partial \gamma}. \tag{A.39}$$

Bibliography

Avrim L. Blum and Pat Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1-2):245–271, 1997.

Jim Bowring, Alessandro Orso, and Mary Jean Harrold. Monitoring deployed software using software tomography. In Matthew B. Dwyer, editor, *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE-02)*, volume 28, 1 of *SOFTWARE ENGINEERING NOTES*, pages 2–9. ACM Press, 2002.

Lisa Burnell and Eric Horvitz. Structure and chance: melding logic and probability for software debugging. *Communications of the ACM*, 38(3):31–41, 57, March 1995.

Fan R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

Sebastian Elbaum and Madeline Hardojo. Deploying instrumented software to assist the testing activity. In RAMSS 2003 RAMSS 2003 [2003], pages 31–33.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37 of *5*, pages 234–245. ACM Press, 2002.

Jerome H. Friedman and Jacqueline J. Meulman. Clustering objects on subsets of variables. *Journal of the Royal Statistical Society*, 4:815–849, 2004.

Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654. IEEE Computer Society, 2004.

Kenny C. Gross, Scott McMaster, Adam Porter, Aleskey Urmanov, and Lawrence G. Votta. Proactive system maintenance using software telemetry. In RAMSS 2003 RAMSS 2003 [2003], pages 24–26.

Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, March 2003.

Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 291–301. ACM Press, 2002.

J. A. Hartigan. Direct clustering of a data matrix. *Journal of the American Statistical Association*, 67(337):123–129, 1972.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer–Verlag, 2001.

David Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-02)*, volume 37 of *1*, pages 58–70. ACM Press, 2002.

Jean-Baptiste Hiriart-Urruty and Claude Lemarechal. *Convex Analysis and Minimization Algorithms*, volume II. Springer–Verlag, 1993.

Tommi Jaakkola and Michael Jordan. Variational probabilistic inference and the QMR-DT database. *Journal of Artificial Intelligence Research*, 1:1–15, 1999.

Nathalie Japkowicz and Shaju Stephen. The class imbalance problem: a systematic study. *Intelligent Data Analysis Journal*, 6(5), November 2002.

Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 11th USENIX Security Symposium, USENIX*, August 2004.

Erich L. Lehmann. *Testing Statistical Hypotheses*. John Wiley & Sons, 2nd edition, 1986.

Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *ACM SIGPLAN PLDI 2003*, 2003.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Public deployment of cooperative bug isolation. In *Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04)*, pages 57–62, Edinburgh, Scotland, May 24 2004.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN PLDI 2005*, 2005.

Benjamin Robert Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, December 2004.

Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: Statistical model-based bug localization. In *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)*, 2005.

Sara C. Madeira and Arlindo L. Oliveira. Biclustering algorithms for biological data analysis: a survey. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(1):24–45, January 2004.

Mario Marchand and John Shawe-Taylor. The set covering machine. *Journal of Machine Learning Research*, 3:723–746, 2002.

Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS 14*, 2002.

Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 128–137. ACM Press, 2003.

Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 465–477. IEEE Computer Society, 2003.

RAMSS '03: The 1st International Workshop on Remote Analysis and Measurement of Software Systems, May 2003.

Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In ACM, editor, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pages 76–85, New York, NY 10036, USA, 2003. ACM Press.

Robert Tibshirani, Trevor Hastie, Mike Eisen, Doug Ross, David Botstein, and Pat Brown. Clustering methods for the analysis of DNA microarray data. Technical report, Department of Health Research and Policy, Department of Genetics and Department of Biochemistry, Stanford University, October 1999.

L. G. Valiant. A theory of the learnable. *Communications of the Association of Computing Machinery*, 27(11):1134–1142, November 1984.

Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In Douglas H. Fisher, editor, *Proceedings of ICML-97, 14th International Conference on Machine Learning*, pages 412–420, Nashville, US, 1997. Morgan Kaufmann Publishers, San Francisco, US.

Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, Charleston, South Carolina, November 2002.