How Bayesians Debug

Chao Liu
Dept. of Computer Science
University of Illinois-UC
Urbana, IL 61801
chaoliu@cs.uiuc.edu

Zeng Lian
Dept. of Mathematics
Brigham Young University
Provo, UT 84602
zenglian@math.byu.edu

Jiawei Han
Dept. of Computer Science
University of Illinois-UC
Urbana, IL 61801
hanj@cs.uiuc.edu

Abstract

Manual debugging is expensive. And the high cost has motivated extensive research on automated fault localization in both software engineering and data mining communities. Fault localization aims at automatically locating likely fault locations, and hence assists manual debugging. A number of fault localization algorithms have been developed in recent years, which prove effective when multiple failing and passing cases are available. However, we notice what is more commonly encountered in practice is the two-sample *debugging problem, where only one failing and one passing cases are available. This problem has been either overlooked or insufficiently tackled in previous studies.*

In this paper, we develop a new fault localization algorithm, named BayesDebug*, which simulates some manual debugging principles through a Bayesian approach. Different from existing approaches that base fault analysis on multiple passing and failing cases,* BayesDebug *only requires one passing and one failing cases. We reason about why* BayesDebug *fits the two-sample debugging problem and why other approaches do not. Finally, an experiment with a real-world program* `grep`*-2.2 is conducted, which exemplifies the effectiveness of* BayesDebug*.*

1 Introduction

Software reliability is one of the top concerns in modern industry. According to a report from the National Institute of Standards and Technology (NIST), software faults (*i.e.*, bugs or errors) cost the U.S. economy an estimated 59.5 billion dollars annually, or approximately 0.6% of the GDP in 2002. But on the other hand, fault elimination through manual debugging is notoriously expensive. As computers become more and more powerful, one may naturally wonder *whether computers can be used to automate the debugging process, or at least to some extent.*

This problem, called *fault localization*, has been extensively studied in both software engineering community [6, 8, 15] and data mining community (though less extensively) [9, 10, 16]. Fault localization aims at *localizing* the underlying faults through automated analysis. With a localization result (*e.g.*, what code regions likely contain the fault), a developer can prioritize the source code examination, and hence debug in a guided way. But one needs to keep in mind that although fault localization is sometimes called *automated debugging*, it is nevertheless a means of debugging assistance. Human developers still need to *precisely* locate the fault and finally fix it.

Due to the practical importance, many fault localization algorithms have been developed in the past decade [6, 8, 13, 15, 16]. Among them, statistical debugging, represented by the algorithms Liblit05 [6] and Sober [8], is shown among the most effective [7]. In statistical debugging, the faulty program is first instrumented with many program predicates. A predicate is a proposition about any program property, *e.g.*, "`idx <= LENGTH`". When a predicate is *instrumented* at a certain piece of code, it gets evaluated (as either `true` or `false`) every time the associated code is executed. At run time, the evaluation history for each predicate is recorded, and finally each execution is profiled as a vector of predicate evaluations, with each dimension for one instrumented predicate.

When n *passing cases* (*i.e.*, execution with correct outputs) and m *failing cases* (*i.e.*, execution with incorrect outputs) are available, statistical debugging algorithms, taking as inputs the profiled evaluation vectors, rank all instrumented predicates according to their calculated likelihood to be fault-relevant. In consequence, source code corresponding to the most relevant predicates is regarded the suspicious fault location. In previous study [6, 8], quality localization results have been

observed with both LIBLIT05 and SOBER when n and m are reasonably large, *e.g.*, in the tens or hundreds. For example, a number of previously unreported faults are found in bc-1.06, EXIF-0.6.9 and Rhythmbox-0.6.5, which have tens of thousands of lines of code. But, besides success stories, it is also noticed that when n and m decrease, the localization quality (not surprisingly) degrades.

Meanwhile, we also notice that in practice usually not many executions are available, and even worse, it is very likely that only *one* failing case is available. For example, when a failing case is encountered, a developer rarely bothers with finding more cases before debugging. This puts forward the *two-sample* debugging problem: *Given one failing execution and one passing execution, how can we obtain quality fault localization*? Here, a sample refers to a *labelled* execution, and an execution is "labelled" if it has been identified as either passing or failing. For automated fault localization, at least one passing case is needed, although a human developer may not need it.

In comparison with the situation when multiple passing and failing cases are available, the two-sample problem is more challenging and practically valuable. In the first place, with only two samples, existing statistical approaches are no longer effective due to the small sample size (see details in Section 4). Secondly, the two-sample debugging problem is more commonly encountered in practice because labelled executions are expensive to collect. Specifically, because of the distinct functionality of each program, developers need to prepare exclusive test cases for a given program. Also, developers need to label each execution as either passing or failing themselves. Although some tools, like Korat [1], can help expedite test preparation, critical manual work is still unavoidable, especially for labelling executions. Finally, as we know, developers are usually reluctant to find more failing cases once one is encountered.

In this paper, we present our investigation of the two-sample debugging problem. In the pursuit of a proper approach, we notice that the process of manual debugging exhibits some principles that could be potentially exploited by automated methods. To be specific, when a failing execution is encountered, the developer will trace it, and contrast it against what it *should* be. If "what it should be" is considered as a passing execution, the manual debugging process just illustrates how to tackle the two-sample problem. A perusal of the above process reveals the following principle that successful manual debugging relies on: The developer has a clear notion of correctness (*i.e.*, what the execution should be) and a notion of incorrectness (*i.e.*, how the failing case executes); then the developer contrasts the notion of incorrectness against that of correctness, and the place where the two notions diverge is regarded as a possible fault location.

While inviting, a literal implementation of the above principle in computers is extremely hard. In consequence, several tradeoffs are made in this study. First, because no knowledge of the correct program behavior is known[1], the notion of correctness is automatically derived from the given passing execution. Although the notion so derived could be far from what human developers think during debugging, it is, as we will see, nevertheless useful. Meanwhile, this derivation also facilitates a higher level of automation. Secondly, notions of correctness and incorrectness are encoded based on predicate evaluations, but not in the form of artificial intelligence. Finally, notions are only contrasted at the end of the executions, rather than at certain suspicious locations *during* the executions as well, because these locations are exactly what a fault localization algorithm needs to find.

Following the above principle and tradeoffs, a new statistical debugging algorithm is developed in this study. As the first step, notions of correctness and incorrectness are formed based on the Bayes' Theorem,

$$Pr(\theta|X) = \frac{Pr(X|\theta)Pr(\theta)}{Pr(X)}. \tag{1}$$

Specifically, let θ ($\theta \in [0,1]$) represent the probability for a predicate P to be evaluated as `true` in passing executions. Then $Pr(\theta)$ is the *prior notion* of correctness about P before any evaluations are observed. With no prior belief, $Pr(\theta)$ is a uniform prior $U(0,1)$. Let $\mathbf{X_p}$ stand for the evaluations of P observed in the passing execution, then $Pr(\theta|\mathbf{X_p})$ is the *posterior* notion of correctness after $\mathbf{X_p}$ is observed. By treating $\mathbf{X_p}$ as a series of independent Bernoulli trials, a simple calculation (based on Eq. (1)) shows that the posterior $Pr(\theta|\mathbf{X_p})$ is a Beta distribution. This distribution encodes one's belief in where, and in what probability, θ should be between 0 and 1 after observing the correct execution. Similarly, if $\mathbf{X_f}$ is the evaluations in the failing execution, $Pr(\theta|\mathbf{X_f})$ embodies one's posterior belief in θ from the failing execution. In this way, the notions of correctness and incorrectness are mathematically represented.

Once the closed form of $Pr(\theta|\mathbf{X_p})$ and $Pr(\theta|\mathbf{X_f})$ is obtained, notion contrast is fairly straightforward although the actual mathematical derivation can be quite involved. As expected, the Kullback-Leibler divergence

[1] If such knowledge is indeed available, model checking should be a better choice. In general, such knowledge is expensive to specify because non-trivial manual work is required.

(a.k.a. KL divergence) is used to quantify the divergence between $Pr(\theta|\mathbf{X_f})$ and $Pr(\theta|\mathbf{X_p})$. Intuitively, the larger the divergence, the more likely the predicate P is fault-relevant. Finally, all instrumented predicates are ranked in the decreasing order of the divergence, and the source code corresponding to the top predicates is expected to be examined first. As this algorithm forms the notion of incorrectness and correctness based on the Bayes' Theorem, it is named BayesDebug.

In summary, we make the following contributions in this paper.

1. We identify the two-sample debugging problem in fault localization, which is both intellectually challenging and practically valuable. This problem has been overlooked or insufficiently tackled in previous studies, and this paper presents the first in-depth investigation of this problem.
2. Inspired by manual debugging experiences, we develop an automated fault localization algorithm, named BayesDebug, which simulates some manual debugging principles based on the Bayes' Theorem. Our experiment clearly demonstrates its effectiveness over existing approaches. Moreover, some reasonings about why BayesDebug is effective for the two-sample debugging problem is also provided.
3. This study provides yet another success story that exemplifies the strength of data mining in software engineering. In particular, this paper demonstrates how Bayesian reasonings can help developers localize software faults. To the best of our knowledge, this is the first piece of work on Bayesian debugging.

The rest of the paper is organized as follows. In Section 2, we first provide an example to illustrate how the notion of correctness and incorrectness is formed through a Bayesian approach. The development of BayesDebug and related proofs are presented in Section 3. We reason why BayesDebug, but not the existing algorithms, is suitable for the two-sample debugging problem in Section 4. Detailed experimental study and comparison are presented in Section 5, which is followed by discussion about related and future work in Section 6. Finally, Section 7 concludes this study.

2 An Illustrative Example

In this section, we illustrate the basic ideas of BayesDebug through a simple example. We hope this example could clarify the meaning of notions, and why *notion divergence* can be related to potential fault locations.

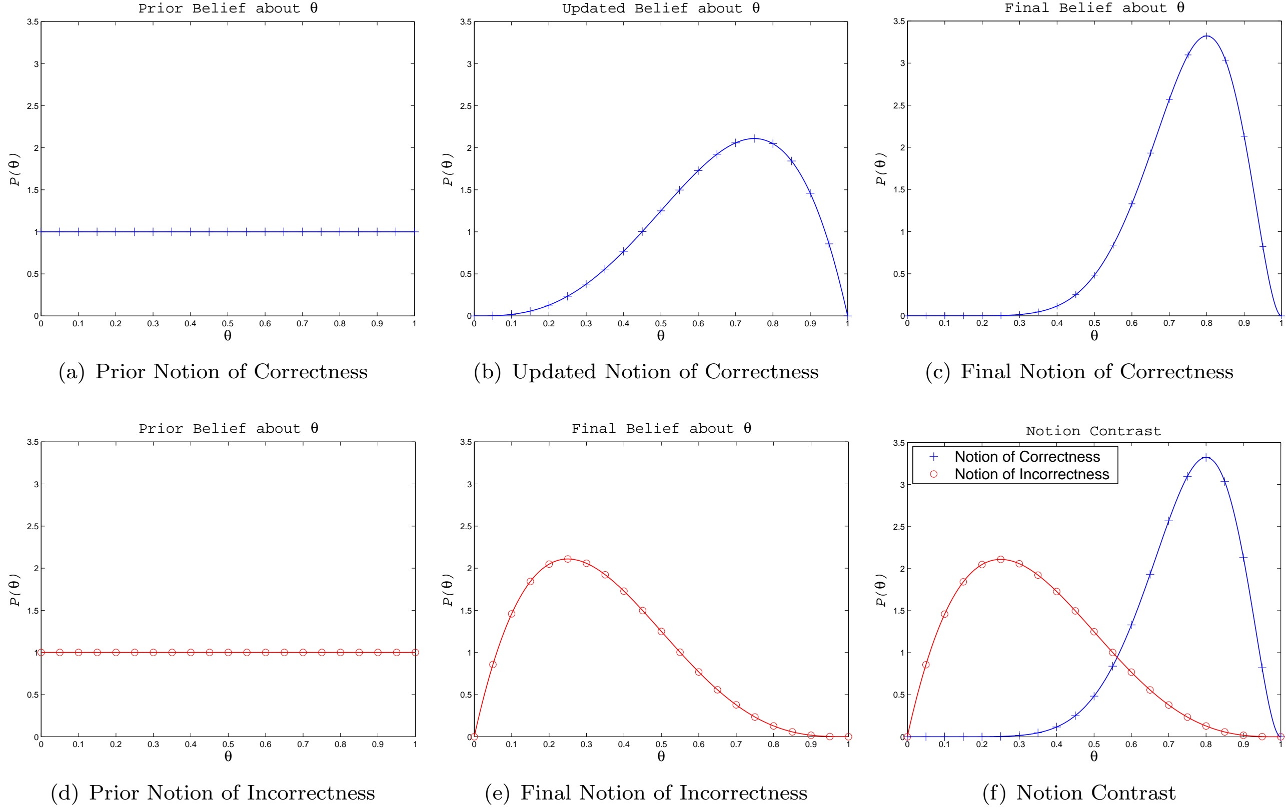
Suppose we now encounter a failing execution f on a program $\mathcal{P}$, together with one passing execution p. Let P denote one of the many instrumented predicates, and imagine a Bayesian sits right beside the predicate P, monitoring the evaluations of P during the two executions. Let θ be a random variable standing for the probability of P being evaluated as `true` in every evaluation, we now illustrate how the Bayesian updates its belief in θ as the program executes.

Before an execution starts, the Bayesian is totally uncertain about what value θ could be between 0 and 1 in the passing and failing executions. The uncertainty is represented by a uniform prior as shown in Figures 1(a) and 1(d). Markers are for legibility with black/white printing. Then, let the execution p start. At some time in the execution, the Bayesian observes three `true` and one `false` executions of the predicate P. Based on this observation, the Bayesian will update its belief accordingly: rather than being any value between 0 and 1 with equal probability, θ is more likely to be larger than 0.5. Figure 1(b) exactly depicts this posterior, and it is a Beta distribution. Suppose five more `true` and one more `false` evaluations of P are observed until the end of p, the posterior belief of the Bayesian in θ is shown in Figure 1(c), which represents the notion of correctness held by the Bayesian on the evaluations of predicate P. Similarly, suppose P is evaluated three times as `false` and once as `true` in f, the notion of incorrectness is shown Figure 1, another Beta distribution.

As Figures 1(c) and 1(e) represent the Bayesian's belief about θ in passing and failing executions respectively, we then need to calculate the divergence between them. The contrast is shown in Figure 1(f), and intuitively larger divergence implies more likelihood for predicate P to be fault-relevant because of their divergent evaluation patterns. A mathematical derivation of the divergence is presented in Section 3. In the end, a fault-relevance score is calculated for each instrumented predicate, and source code corresponding to the most relevant predicates is taken as the possible fault location.

3 Bayesian Debugging

In this section, we discuss in detail the design of BayesDebug. In Section 3.1, we first describe what predicates are instrumented in this study. Then we elaborate on how the notions of correctness and incorrectness are formed, and how they are contrasted in Sections 3.2 and 3.3, respectively. Finally, Section 3.4 addresses some related issues to BayesDebug.



(a) Prior Notion of Correctness (b) Updated Notion of Correctness (c) Final Notion of Correctness

(d) Prior Notion of Incorrectness (e) Final Notion of Incorrectness (f) Notion Contrast

Figure 1. An Illustrative Example of BayesDebug

3.1 Predicate Instrumentation

In order to profile executions, a subject program is first instrumented with predicates. In this study, two kinds of predicates, **branch** and **call**, are instrumented in subject programs. Specifically, for each branch B, two predicates $B = \texttt{true}$ and $B = \texttt{false}$ are instrumented; and for each function call site C, six predicates are instrumented: $C \leq 0$, $C > 0$, $C = 0$, $C \neq 0$, $C < 0$, and $C \geq 0$. By virtue of a compiler frontend, the instrumentation is fully automated.

For each predicate P, the corresponding source code location L is called its *instrumentation site*. Equivalently, we say the predicate P *points to* L. During an execution, a predicate P is *evaluated* (as either `true` or `false`) when its instrumentation site is executed. At the end of the execution, the evaluation history of each predicate is dumped out for subsequent analysis. Because this study treats each predicate equally and independent of each other, the following discussion applies to any predicate P, and the given passing and failing executions in the two-sample debugging problem are denoted by p and f, respectively.

3.2 Notion Formation

Let X be a random variable representing the evaluation of predicate P in an execution. $X = 1$ if P evaluates `true`, and 0 if `false`. With independence assumption between evaluations, X conforms to a Bernoulli distribution with head-probability θ, *i.e.*,

$$X \sim f(X|\theta) = \theta^X(1-\theta)^{(1-X)}, \quad \theta \in [0,1].$$

Then, we are interested in whether θ is the same between the failing execution f and the passing execution p, and if not, how large the difference is. Intuitively, the larger the difference, the more likely the predicate P is fault-relevant.

Suppose the predicate P is evaluated n times, which are denoted by $\mathbf{X} = (X_1, X_2, \cdots, X_n)$. The following lemma indicates that given a Beta distribution prior for θ, the posterior about θ is also a Beta distribution after $\mathbf{X}$ is observed.

Lemma 1. *Given a Beta distribution prior for θ, i.e., $f(\theta|\Phi) = Beta(\alpha, \beta)$, where Φ denotes prior knowledge, and a series of observations $\mathbf{X} = (X_1, X_2, \cdots, X_n)$ from the Bernoulli(θ) distribution,*

the posterior about θ after $\mathbf{X}$ *is observed is*

$$f(\theta|\mathbf{X},\Phi) = Beta(\alpha + \sum_{i=1}^{n} X_i,\ \beta + n - \sum_{i=1}^{n} X_i).$$

Proof. When X_1 is observed, the posterior of θ is

$$\begin{aligned} f(\theta|X_1,\Phi) &= \frac{f(X_1|\theta,\Phi)f(\theta|\Phi)}{f(X_1|\Phi)} \\ &= \frac{\theta^{X_1}(1-\theta)^{1-X_1}\frac{1}{\mathcal{B}(\alpha,\ \beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}}{\int_0^1 \theta^{X_1}(1-\theta)^{1-X_1}\frac{1}{\mathcal{B}(\alpha,\ \beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}d\theta} \\ &= \frac{1}{\mathcal{B}(\alpha+X_1,\ \beta+1-X_1)}\theta^{\alpha+X_1-1}(1-\theta)^{\beta-X_1} \\ &= Beta(\alpha+X_1,\ \beta+1-X_1) \end{aligned}$$

where $\mathcal{B}(\alpha,\beta)$ is the beta function,

$$\mathcal{B}(\alpha,\beta) = \int_0^1 x^{\alpha-1}(1-x)^{\beta-1}dx.$$

Therefore, the posterior about θ after X_1 is observed is $Beta(\alpha+X_1,\ \beta+1-X_1)$. Through a simple induction on X_2, X_3, $\cdots$, we know the posterior about θ after $\mathbf{X}$ is observed is

$$f(\theta|\mathbf{X},\Phi) = Beta(\alpha + \sum_{i=1}^{n} X_i,\ \beta + n - \sum_{i=1}^{n} X_i).\quad \square$$

The proof of Lemma 1 describes how the posterior about θ gets updated every time the predicate P is evaluated. At the beginning, with no prior belief in θ, a uniform prior should be assigned to θ, *i.e.*, $f(\theta|\Phi) = 1/\theta,\ \theta \in [0,1]$. Because the uniform distribution is a special Beta distribution with parameters $\alpha = \beta = 1$, Lemma 1 immediately gives the posterior belief for each predicate after either the passing execution p or the failing execution f is observed.

Specifically, suppose $\mathbf{X_p} = (X_1, X_2, \cdots, X_n)$ and $\mathbf{X_f} = (X'_1, X'_2, \cdots, X'_m)$ represent the evaluations of a predicate P in p and f respectively, then the posterior of P from the passing execution p is

$$\theta_p \sim f(\theta|\mathbf{X_p}) = Beta(1 + \sum_{i=1}^{n} X_i,\ n+1-\sum_{i=1}^{n} X_i)$$

and similarly, the posterior of P from f is

$$\theta_f \sim f(\theta|\mathbf{X_f}) = Beta(1 + \sum_{i=1}^{m} X'_i,\ m+1-\sum_{i=1}^{m} X'_i).$$

For clear notation, we have saved Φ from the above expressions.

Go back to Figure 1. The distributions in Figures 1(c) and 1(e) are $Beta(9,3)$ and $Beta(2,4)$ respectively. In the next subsection, we mathematically quantify the divergence between $f(\theta|\mathbf{X_p})$ and $f(\theta|\mathbf{X_f})$.

3.3 Notion Contrast

Given the closed forms of $f(\theta|\mathbf{X_p})$ and $f(\theta|\mathbf{X_f})$, which are essentially two distributions, the Kullback-Leibler divergence is a natural choice to quantify the divergence. In its original form, given two continuous distributions $p(x)$ and $q(x)$, the KL-divergence $KL(p \mid\mid q)$ is defined as

$$KL(p \mid\mid q) = \int_{-\infty}^{+\infty} p(x) \ln \frac{p(x)}{q(x)} dx. \qquad (2)$$

With the closed form of $f(\theta|\mathbf{X_p})$ and $f(\theta|\mathbf{X_f})$, it is possible to obtain an analytical form of the KL-divergence, and an analytical form, if obtained, could circumvent the instability and inefficiency of the numeric computation of integrals.

For easy notation, let

$$a = 1 + \sum_{i=1}^{n} X_i, \qquad b = n+1-\sum_{i=1}^{n} X_i$$
$$c = 1 + \sum_{i=1}^{m} X'_i, \quad d = m+1-\sum_{i=1}^{m} X'_i,$$

then the KL-divergence is calculated with $p(x) = Beta(c,d)$ and $q(x) = Beta(a,b)$. But, before plugging them into Eq. (2), we first give the following Lemma, which is essential for computing the KL-divergence between two Beta distributions.

Lemma 2.

$$\int_0^1 \theta^{c-1}(1-\theta)^{d-1}\ln(\theta)d\theta = \frac{\Gamma(c)\Gamma(d)}{\Gamma(c+d)}[\Psi(c)-\Psi(c+d)],$$

where c and d are positive integers, $\Gamma(x) = \int_0^{+\infty} t^{x-1}e^{-t}dt$, *and* $\Psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$.

Proof. Sketch of proof is in the appendix. $\square$

With Lemma 2, the following theorem gives an analytical form of the divergence between $f(\theta_f|\mathbf{X_f})$ and $f(\theta_p|\mathbf{X_p})$.

Theorem 1 (Divergence Between Posteriors)**.** *The KL-divergence between $f(\theta_p|\mathbf{X_f})$ and $f(\theta_f|\mathbf{X_p})$, denoted as $KL(\theta_f \mid\mid \theta_p)$, is*

$$\ln\frac{\mathcal{B}(a,b)}{\mathcal{B}(c,d)} + (c-a)[\Psi(c)-\Psi(c+d)] + (d-b)[\Psi(d)-\Psi(c+d)],$$

where $\Gamma(x) = \int_0^{+\infty} t^{x-1}e^{-t}dt$, *and* $\Psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$.

Proof. Sketch of proof is in the appendix. $\square$

Finally, we can take $KL(\theta_f||\theta_p)$ as the fault-relevance score for the predicate P, and calculate it according to THEOREM 1. Then all predicates are ranked in decreasing order, and top predicates are regarded more relevant to the fault. Since Bayesian modelling of θ_p and θ_f is used, this fault localization algorithm is named BAYESDEBUG.

3.4 Computation and Quality Sensitivity

In this subsection, we discuss some related issues to the BAYESDEBUG algorithm. First, the computation efficiency. As shown in THEOREM 1, the analytical form of the KL-divergence is formidably complex. For example, computing the beta function $\mathcal{B}$ requires integrals, and the digamma function Ψ further involves derivatives of integrals. However, thanks to the development in numeric computation of special functions [12], both $\mathcal{B}$ and Ψ can be computed within $O(1)$ time. Therefore, once the executions p and f exit, the fault-relevance score for each predicate can be calculated within $O(1)$ time. Finally, a sorting of all instrumented predicates according to the scores is needed. Therefore, suppose k predicates are instrumented, the computation complexity of BAYESDEBUG is $O(k \log(k))$.

Secondly, as one may have noticed, the localization quality could be sensitive to the given passing and failing cases. Intuitively, the failing execution f should be *similar* to the given passing one in order to obtain quality localization result. If it is, the evaluation pattern of most predicates would be roughly the same, and in consequence, real fault-relevant predicates will be put at the top. So far, no formal justification exists for the "similarity implies quality result" principle although some previous work on fault localization proposes similar conjectures [13, 15]. Some experimental results on this will be presented in Section 5.

4 Methodology Comparison

In this section, we compare BAYESDEBUG, from the methodology point of view, with three well-established fault localization algorithms, namely, SOBER [8], LIBLIT05 [6] and Delta Debugging (abbreviated as DELTADEBUG) [15]. Specifically, we discuss the methodology difference between BAYESDEBUG and each of them. This discussion illustrates why BAYESDEBUG is suitable for the two-sample debugging problem, and why the other three algorithms are not. Meanwhile, because we will compare BAYESDEBUG with them through experiments in the next section, this section also serves as an introduction to the three algorithms.

4.1 Comparison with SOBER

SOBER is a statistical debugging algorithm recently proposed by Liu et al. [8]. For each predicate P, SOBER models each evaluation as an independent Bernoulli trial with head-probability θ. Furthermore, SOBER regards that there exist two distributions $f(\theta|\Phi_p)$ and $f(\theta|\Phi_f)$ that govern the head probability θ for passing and failing executions respectively. For each passing (or failing) execution, the `true` evaluation ratio of P, calculated by dividing the number of `true` evaluations over the total number of evaluations, is taken as an observation from $f(\theta|\Phi_p)$ (or $f(\theta|\Phi_f)$ correspondingly). When n passing and m failing executions are available, the n and m observations constitute one sample from $f(\theta|\Phi_p)$ and $f(\theta|\Phi_f)$ respectively. Finally, the divergence between $f(\theta|\Phi_p)$ and $f(\theta|\Phi_f)$ is quantified in a similar way to the two-sample hypothesis testing. Readers interested in the details are referred to [7].

The above description reveals some similarities between SOBER and BAYESDEBUG. First, both methods model predicate evaluations as independent Bernoulli trials. Secondly, both methods base fault localization on the head probability θ. Moreover, in both methods, two probability distributions are used to describe the uncertainty of θ. Finally, both methods relate the probability divergence to the fault-relevance score. Therefore, at the first glance, BAYESDEBUG follows the route of SOBER.

However, a perusal of the two algorithms exposes important differences. First, BAYESDEBUG and SOBER approach the problem of fault localization in a *Bayesian* and a *Frequentist* way, respectively. This difference manifests itself in the meaning of the distribution of θ. In SOBER, $f(\theta|\Phi_p)$ is a fixed (although unknown) distribution that governs the head probability of predicate P in passing executions. In contrast, the $f(\theta_p|\mathbf{X_p})$ in BAYESDEBUG is the posterior belief in θ's distribution after the evaluations $\mathbf{X_p}$ are observed from *the* passing execution. $f(\theta_p|\mathbf{X_p})$ gets updated every time P is evaluated during the execution p while $f(\theta|\Phi_p)$ is regarded fixed throughout all passing executions.

Secondly, while $f(\theta_p|\mathbf{X_p})$ is obtained from *one* execution in BAYESDEBUG, SOBER essentially needs *multiple* executions to estimate $f(\theta|\Phi_p)$. For this reason, SOBER has a hard time to handle the two-sample debugging problem because only one observation is available for the estimation of $f(\theta|\Phi_p)$. In contrast, since each evaluation is taken as an independent observation, BAYESDEBUG actually collects multiple observations from each execution. Finally, the two algorithms also fundamentally differ in their approaches to quantifying the distribution divergence. SOBER measures the di-

vergence through a hypothesis testing approach, while BayesDebug simply gives an analytical form of the divergence. Therefore, BayesDebug is fundamentally different from Sober although they look similar at the first glance.

In previous study, Sober is observed to be the most effective debugging algorithm, evaluated when multiple failing and a great number of passing executions are available. However, for the two-sample debugging problem, Sober may not be able to correctly estimate the θ distribution in both failing and passing executions (due to high estimation variance). In consequence, the divergence measured through hypothesis testing is also less credible. Experimental comparison between BayesDebug and Sober is presented in Section 5.

4.2 Comparison with Liblit05

Besides Sober, Liblit05 is another representative of statistical debugging algorithms [6]. Different from both Sober and BayesDebug, Liblit05 sifts fault-relevant predicates by examining how more likely an execution fails when a predicate evaluates `true` than when the predicate is ever evaluated as either `true` or `false`. Specifically, Liblit05 defines the following two probabilities for each predicate P,

$$Context(P) = Pr(Fail|P\ evaluated), \quad (3)$$
$$Failure(P) = Pr(Fail|P\ evaluated\ \texttt{true}), \quad (4)$$

and takes the probability difference

$$Increase(P) = Failure(P) - Context(P), \quad (5)$$

as one of the two key components of P's fault-relevance score. The other component is invalid in the two-sample debugging problem setting, and is hence ignored in this discussion.

When multiple passing and failing executions are available, $Context(P)$ is estimated by the ratio of failing executions among all executions in which the predicate P is ever evaluated. Similarly, $Failure(P)$ is estimated by the ratio of failing executions among all executions in which the predicate P is ever evaluated `true`. $Increase(P)$ is then the difference between the two estimations. Because only two observations (one from the failing and one from the passing execution) are available for estimation, we envision that Liblit05 will also have a hard time as Sober for the two-sample debugging problem. This is confirmed in the experiments in Section 5.

4.3 Comparison with Delta Debugging

Finally, we would like to compare BayesDebug with DeltaDebug, which is proposed by Zeller in [15]. Different from both Sober and Liblit05, DeltaDebug rightly fits the two-sample debugging problem because it literally contrasts one failing execution against a passing one. The contrast is carried on the *memory graphs* that are induced from the two executions. Intuitively, a memory graph is a graphic representation of the reference relationship between program variables. By systematically exchanging the content of the two graphs, DeltaDebug finally locates those failure-inducing variables, and these variables and their associated source code are regarded as a fault localization result.

The DeltaDebug service is available online at `www.askigor.com`, where a command-line utility is also ready for offline use. DeltaDebug is effective if the underlying fault incurs some memory abnormality, like access violations or reference circles that should not exist. However, if the fault manifest no abnormality at the memory level, DeltaDebug is no longer effective. For example, most logic faults would only result in erroneous outputs without program crashes. For such cases, predicate-based debugging algorithms are more effective than DeltaDebug, as we will see in the following section.

5 Experiment Results

In this section, we evaluate the effectiveness of BayesDebug with experiments. We first describe the subject program and the seeded faults in Section 5.1. Then in Sections 5.2 and 5.3, we examine the effectiveness of BayesDebug, and compare it with Sober, Liblit05 and DeltaDebug. The sensitivity of localization quality is studied in Section 5.4, where the computation cost is also briefly discussed. Finally, Section 5.5 discusses some threats to validity, and closes this experimental study.

5.1 Subject Program and Seeded Faults

We choose grep-2.2 as the subject program in this study. The grep program searches one or more input files for lines containing a match to a specified pattern, and prints out matching lines. It has 9,745 lines of C code, as measured by the SLOCCount tool[2]. Two logic faults are manually seeded in the source code, and they are shown in Figure 2.

The first fault (on the left) is an "off-by-one" error: an expression "+1" is appended to line 553 in the grep.c file. The second fault (on the right) is a "subclause-missing" error. The subclause `(lcp[i] == rcp [i])` at line 2270 in file dfa.c is commented out.

[2] Available at http://www.dwheeler.com/sloccount/

```
    static int grep(int fd)
    {
     ...
541  for( ; ; )
542  {
      ...
548   lastnl = bufbeg;
549   if (lastout)<-------------------- P2
550     lastout = bufbeg;
551   if (buflim - bufbeg == save)
552     break;
553   beg = bufbeg + save - residue + 1; /* fault 1 */
554   for(lim = buflim; lim > beg && lim[-1] != '\n'; --lim)
555     ;
      ...
574   if (beg != lastout)<------------------ P1
575     lastout = 0;
576   save = residue + lim - beg;
      ...
580  }
     ...
587  return nlines;
588 }
```

Fault 1: An off-by-one error in grep.c

```
      static char ** comsubs(char* left, char* right)
      {
       ...
2264   for(lcp = left; *lcp != '\0'; ++lcp)
2265   {
2266     len = 0;
2267     rcp = index(right, *lcp);                       P3
2268     while (rcp != NULL)
2269     {
2270       for (i = 1; lcp[i] != '\0'/* && lcp[i] == rcp[i] */; ++i) /* fault 2 */
2271         continue;
2272       if (i > len)
2273         len = i;
2274       rcp = index(rcp + 1, *lcp);
2275     }
2276     if (len == 0)
2277       continue;
2278     if ((cpp = enlist(cpp, lcp, len)) == NULL)
2279       break;
2280   }
2281   return cpp;
2282 }
```

Fault 2: A subclause-missing error in dfa.c

Figure 2. Two Seeded Faults

Although these two faults are manually injected, they do mimic realistic logic errors. Explicitly, when developers are obscure about corner conditions, logic errors like "off-by-one" or "subclause-missing" may sneak in. Because logic errors, like these two, generally do not incur segmentation faults, they are usually harder to debug than those with program crashes. In the next subsection, we illustrate how BayesDebug helps developers find these two faults separately.

5.2 Effectiveness of BayesDebug

We now examine the effectiveness of BayesDebug in localizing the two seeded faults respectively. According to the instrumentation schema described in Section 3.1, 3464 **branch** and 1404 **call** predicates are instrumented in the `grep` subject. The BayesDebug is implemented within Matlab, and all experiments are carried out on a Pentium-IV Linux machine running Fedora Core 2.

Table 1. Top Two Predicates for Fault 1

	Predicate Content	Location
P_1	`(beg != lastout)=true`	Line 574 of `grep.c`
P_2	`(lastout)=true`	Line 549 of `grep.c`

We first enable the first fault only, and construct two test cases such that one is passing and the other is failing. We denote them as p_1 and f_1 respectively. After applied to the execution traces of p_1 and f_1, BayesDebug generates a ranking of all instrumented predicates, with the top two predicates shown in Table 1. For easy reference, the two predicates are marked at their instrumentation sites in Figure 2.

As we can see, the predicates P_1 and P_2 point to the faulty function for the first fault. Especially, the predicate P_2 is only 4 lines above the real fault location. Now let us pretend to be a developer encountering the failing case f_1, and explore how the top predicates can help us find the fault.

Given the top-ranked predicates, it is natural to ask why they are ranked high. We find that P_1 is evaluated as `true` for ten times and none as `false` in f_1, but 11 times as `false` and none as `true` in p_1. As to P_2, it is always evaluated as `true` in p_1 for 14 times, but only once as `true` and ten times as `false` in f_1.

The different evaluation patterns of P_1 in p_1 and f_1 immediately shed some light on the underlying fault. P_1 always being `false` means that the variable `beg` is always equal to `lastout` at line 574 in p_1. However, in f_1, `beg` is always different from `lastout`. For this reason, we may guess that the value of either `beg` or `lastout` or both may deserve special attention in debugging. Moreover, the ten-time `true` evaluations of P_1 also explain the ten-time `false` evaluation of P_2: the variable `lastout` is always reset to 0 at line 575, which immediately causes the `false` evaluation of P_2 at the next iteration. In this sense, the abnormal evaluation pattern of P_2 is only a subsequent symptom from that of P_1, and it provides little new information for debugging. Therefore, we only need to pay attention to P_1, *i.e.*, examining locations where either the variable `beg` or `lastout` (or both) gets assigned.

With this guidance, a developer can place watches on the variables `beg` and `lastout`, and re-execute f_1. With familiarity with the code and program semantics, the developer will notice the assignment to `beg` at line 553, and then locate the fault. In contrast, without BayesDebug, the developer may need to hunt the fault at large, and trace the execution step by step.

After fixing the first fault, let us now turn to the second. We construct another two test cases p_2 and f_2, and a run of BayesDebug on them gives a predicate ranking with the top predicate as "`(lcp[i] != '\0')=true`", which we denote as predicate P_3. This

predicate points to Line 2270 of the `dfa.c` file, and it is the exact location of the second fault. Therefore, a developer can trace the execution of f_2, and put more attention on Line 2270 of the `dfa.c` file. Again, with code familiarity and programming expertise, the fault will be fixed without hunting faults at large.

We notice that for the first fault, the predicates P_1 and P_2 are the most fault-relevant predicates, and they are ranked at the top by BayesDebug. Similarly, for the second fault, the most relevant predicate P_3 is also ranked at the top. In the next subsection, we examine how the other three localization algorithms work under the same setting.

5.3 Effectiveness Comparison

		Liblit05	Sober	BayesDebug
Fault 1	P_1	7th	10th	1st
	P_2	23rd	23rd	2nd
Fault 2	P_3	24th	11th	1st

Table 2. Effectiveness Comparison

We applied both Sober and Liblit05 to the same execution traces for the two faults respectively. And Table 2 shows the effectiveness comparison, in terms of the rank of predicates P_1, P_2 and P_3 in the final predicate ranking. A smaller rank means the predicate is ranked higher in the final ranking.

As we can see, Liblit05 ranks the predicates P_1 and P_2 at the 7th and the 23rd for the first fault, and ranks P_3 at the 24th place for the second fault. The result from Sober is similar: P_1 and P_2 are at the 10th and the 23rd for the first fault, and P_3 is the 11th for the second fault. We also examined the top predicates ranked by both Liblit05 and Sober for the two faults, and found that these top predicates are no enlightening for debugging. In comparison, the most relevant predicates are picked up by BayesDebug for both faults. With fault-relevant predicates deeply buried in the final predicate ranking, the localization results from both Sober and Liblit05 are only of limited utility to developers.

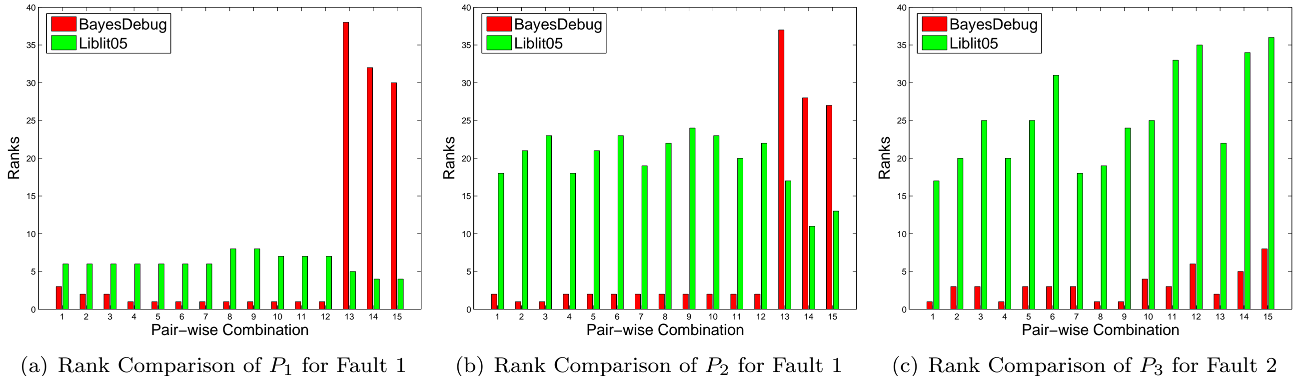
We also subjected DeltaDebug to the two faults separately, and found that DeltaDebug is ineffective for these cases. Specifically, DeltaDebug got choked for the first fault with the passing execution p_1 and the failing execution f_1: It failed to terminate within 48 hours. For the second fault, DeltaDebug immediately completed with executions p_2 and f_2 as inputs, but unfortunately generated an error message: "`Tried all strategies - all failed`", which meant that DeltaDebug failed to find any fault relevant locations for the second fault. According to our discussion in Section 4.3, this result is as expected.

5.4 Quality Sensitivity

As one may expect, the localization quality of BayesDebug or other algorithms in general, could depend on the given failing and passing cases. In order to verify this conjecture, we constructed five passing and five failing cases for each of the two faults, and examined the localization quality of BayesDebug on the 25 pair-wise combinations of passing and failing cases. Because three passing cases resulted in the same predicate ranking with BayesDebug, no matter which of the five failing cases was paired with, two of them were discarded. Therefore, finally 15 combinations were examined for each fault. We also examined the localization quality of both Sober and Liblit05 for the same combinations, and similar results were observed for the two algorithms. In the following, only Liblit05 is compared with BayesDebug.

Figure 3 shows the comparison for the 15 combinations on the two faults. The y-axis is the rank of the predicate in the result ranking, and the x-axis is for the indices of the 15 combinations. Especially, the 15 combinations are arranged with failing cases as the main index. In other words, 1, 2 and 3 are for the combinations of the first failing case with the first, second and third passing cases respectively, and 4, 5 and 6 are for the second failing case with the three passing cases. Figures 3(a) and 3(b) plot the rank comparison of predicates P_1 and P_2 on the first fault, and Figure 3(c) for P_3 on the second fault.

A first look at the three figures reveals that BayesDebug generally has a better localization quality than Liblit05. For most combinations, the fault-relevant predicates are ranked higher by BayesDebug than Liblit05. However, we also observe that for some combinations, BayesDebug ranks the fault-relevant predicates much lower than Liblit05. For example, in Figure 3(a), the predicate P_1 is ranked very low (around or even below 30th) by BayesDebug for the last three combinations. Because P_2 is related to P_1, the same thing is observed for P_2 in Figure 3(b). Moreover, we note that the last three combinations in Figures 3(a) and 3(b) correspond to the combinations of the fifth failing case with the three passing cases. This observation indicates that although quality localization result is obtained by BayesDebug for the twelve pair-wise combinations between the first four failing cases and the three passing cases, when the fifth failing case is given, the localization is poor no matter which of the three passing cases is paired with. This is quite reasonable because even for human developers, not all



(a) Rank Comparison of P_1 for Fault 1 (b) Rank Comparison of P_2 for Fault 1 (c) Rank Comparison of P_3 for Fault 2

Figure 3. Comparison of Quality Sensitivity between BayesDebug and Liblit05

failing executions are equally appropriate for debugging. For example, a long-running failing execution is usually harder to debug than a short one although they two can be due to the same fault.

Meanwhile, we also notice that although BayesDebug gets stuck on the last three combinations for the first fault, Liblit05 actually obtains even better result than the other twelve combinations. This observation shows that there is no absolute "good" pair of failing and passing cases: a pair of executions can be good for one algorithm, but may be meanwhile bad for other algorithms. So far, there are no clear conclusions about what pairs of failing and passing cases are good, and this could continue to be an open problem.

Finally, we briefly report the computation cost for BayesDebug. For the 15 combinations on Fault 1, the running time of 15 times of BayesDebug took 0.2358 seconds, while the running time of Liblit05 was 10.2669 seconds. For the 15 combinations on the second fault, BayesDebug and Liblit05 took 0.2492 and 10.2760 seconds, respectively. We believe that the time advantage of BayesDebug mainly comes from the analytical form of the fault-relevance score (Theorem 1). But since both algorithms end within less than one second for each combination, the time advantage is not significant in practice.

5.5 Threats to Validity

All empirical studies are subjected to some threats to validity, and this study is no exception. In the first place, the two faults in the grep subject program are seeded by our authors although they do mimic realistic "off-by-one" and "subclause-missing" errors. Moreover, this experiment is entirely based on the grep subject program. For this reason, more case studies with real faults are needed to establish the effectiveness of BayesDebug in the future. The second threat to validity comes from the hand-crafted test inputs used in Sections 5.2, 5.3 and 5.4. As shown in Section 5.4, the localization quality of BayesDebug could depend on the given pair of failing and passing cases, and deteriorate significantly when a "bad" pair of test cases is given. We construct those test cases based on our understanding of the code and our design to trigger the seeded faults. Therefore, it could be possible that these cases are proper for BayesDebug. But we also note that the unpleasant results from Sober and Liblit05 should not be attributed to the specific test cases because the two algorithms are in principle short for the two-sample debugging problem, as discussed in Section 4. Finally, Section 5.2 illustrates how a developer finds the seeded faults with the localization result provided by BayesDebug. Ideally, this experiment should be carried out with independent developers rather than our authors. But due to the difficulty (and expenses) of controlled user study, currently most fault localization researches are evaluated by the authors [6, 8, 13, 15].

6 Related Work and Discussion

In this section, we review related work and discuss some potential extensions. First, this study is related to software fault localization, a problem that has been actively pursued in recent years [6, 8, 10, 13, 15]. Depending on whether the program crashes, there are two kinds of software faults: memory-related faults and logic faults. Memory-related faults typically incur access violations that finally result in program crashes. Because crashing scenes can provide valuable debugging information, memory faults are generally easier to debug. In addition, some mature tools, like Valgrind and CCured, can also effectively help developers debug by guarding *each* memory access. Delta Debugging [15] improves over these tools in its ability to find

why a program crashes, rather than merely the sites of memory violations. Logic faults, on the other hand, are generally harder to tackle because there can be no memory abnormality even when the output is incorrect. Because no crashing scene is available, even human developers may find it tricky to debug. The predicate-based fault localization tools, like Liblit05 and Sober, are particularly suitable for finding logic faults, although they are also effective in find memory related errors [6, 8]. Besides the predicate-based approach, Liu et al. also show that some logic faults can be located by mining program control flow graphs that are induced from multiple passing and failing executions [10]. The BayesDebug algorithm developed in this paper shares the principle of predicate-based approach. But different from previous studies that assume the existence of multiple passing and failing executions, BayesDebug is especially designed for the two-sample debugging problem, an important fault localization problem that has been overlooked in previous studies.

Secondly, this work also relates to Bayesian statistics, which is most represented by Bayesian networks in data mining community [3]. Because Bayesian networks can express complex dependencies between attributes and can encode both casual and probabilistic semantics, they are usually adopted where prior knowledge is known and critical. For example, Jaroszewicz and Scheffer discuss how to find *unexpected patterns* from data, where the unexpectedness is defined relative to a Bayesian Network that encodes domain prior knowledge [4]. Besides dependency and domain knowledge representation, Bayesian networks are also known for its usage in classification [2]. However, although related, the BayesDebug algorithm is not an application of Bayesian networks to the fault localization problem. Instead, BayesDebug attempts to simulate human debugging heuristics through a Bayesian approach. Although this simulation could be far from what human developers really think in debugging, the study shows that BayesDebug is nevertheless useful, which echoes the well-known saying that "all models are wrong, but some are useful" (due to George Box). To the best of our knowledge, this is the first piece of work that approaches fault localization in a Bayesian way. We envision that the localization quality of BayesDebug can be further leveraged when dependencies between predicates are encoded in a Bayesian network. Currently, all predicates are treated independent from each other, but they are in fact related.

Finally, this study falls into an emerging application domain in data mining: data mining for software engineering. Previous research indicates that proper mining of software data can produce useful results for software engineers. Livshits et al. apply frequent itemset mining algorithms to software revision history, which uncovers programming rules that developers are expected to conform to [11]. Li et al. apply the CloSpan algorithm [14] to source codes, and successfully locate some copy-paste faults in the Linux system [5]. Liu et al. show that mining program control flow graphs can help developers find logic errors [9, 10]. These cases well exemplify the promise and usefulness of data mining in software engineering, and this study provides yet another example.

7 Conclusions

In this paper, we investigated the two-sample debugging problem, and developed a new fault localization algorithm, BayesDebug, to solve this problem. We compared BayesDebug with three well-known localization algorithms, and the experiment clearly demonstrated the effectiveness of BayesDebug. A few interesting problems remain, which constitute part of our future work.

References

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ACM/SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA'02)*, pages 123–133, 2002.

[2] J. Cheng and R. Greiner. Learning bayesian belief network classifiers: Algorithms and system. In *Proc. of 14th Biennial Conf. of Canadian Society on Computational Studies of Intelligence*, pages 141–151, 2001.

[3] D. Heckerman. Bayesian networks for data mining. *Data Mining and Knowledge Discovery (DMKD)*, 1:79–119, 1997.

[4] S. Jaroszewicz and T. Scheffer. Fast discovery of unexpected patterns in data, relative to a bayesian network. In *Proc. of the 11th ACM SIGKDD Int. Conf. on Knowledge Discovery in Data Mining (KDD'05)*, pages 118–127, 2005.

[5] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proc. 2004 Symp. on Operating System Design and Implementation (OSDI'04)*, 2004.

[6] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Proc. of ACM SIGPLAN 2005 Int. Conf. on Programming Language Design and Implementation (PLDI'05)*, 2005.

[7] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. *IEEE Trans. Softw. Eng. (to appear)*.

[8] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *Proc. of the 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE'05)*, 2005.

[9] C. Liu, X. Yan, and J. Han. Mining control flow abnormality for logic error isolation. In *Proc. 2006 SIAM Int. Conf. on Data Mining (SDM'06)*, 2006.

[10] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *Proc. 2005 SIAM Int. Conf. on Data Mining (SDM'05)*, 2005.

[11] V. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proc. of 13th Int. Symp. on the Foundations of Software Engineering (FSE'05)*, 2005.

[12] D. Lozier and F. Olver. Numerical evaluation of special functions. In *AMS Proceedings of Symposia in Applied Mathematics*, pages 79–125, 1994.

[13] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proc. of 18th IEEE Int. Conf. on Automated Software Engineering*, 2003.

[14] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large databases. In *Proc. of the 3th SIAM Int. Conf. on Data Mining*, 2003.

[15] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. of ACM 10th Int. Symp. on the Foundations of Software Engineering (FSE'02)*, 2002.

[16] A. Zheng, M. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems (NIPS)*, 2004.

Appendix

Proof. (Lemma 2)

With c and d being positives, integral by parts gives

$$\int_0^1 \theta^{c-1}(1-\theta)^{d-1}\ln(\theta)d\theta = \sum_{n=0}^{d-1}(-1)^{n+1}\frac{C_{d-1}^n}{(c+n)^2}.$$

Let

$$f_c(d) = \sum_{n=0}^{d-1}(-1)^{n+1}\frac{C_{d-1}^n}{(c+n)^2},$$

and

$$\begin{aligned} g_c(d) &= \frac{\Gamma(c)\Gamma(d)}{\Gamma(c+d)}\Big(\frac{\Gamma'(c)}{\Gamma(c)} - \frac{\Gamma'(c+d)}{\Gamma(c+d)}\Big) \\ &= (-1)\frac{(c-1)!(d-1)!}{(c+d-1)!}\Big(\sum_{n=0}^{d-1}\frac{1}{c+n}\Big) \end{aligned}$$

because

$$\Gamma'(n+1) = n!\Big(\sum_{i=1}^{n}\frac{1}{i} - \Gamma'(1)\Big) \text{ and } \Gamma(n+1) = n!$$

In the following, we prove that $f_c(d) = g_c(d)\ \forall\, c, d \geq 1$ by induction.

$\forall c(\geq 1) \in \mathbb{N}$, we have

$$\begin{aligned} f_c(2) &= \frac{1}{(c+1)^2} - \frac{1}{c^2} \\ &= (-1)\frac{1}{(c+1)c}\Big(\frac{1}{c} + \frac{1}{c+1}\Big) = g_c(2). \end{aligned}$$

Then suppose $f_c(d) = g_c(d),\ \forall c(> 1) \in \mathbb{N}$,

$$\begin{aligned} f_c(d+1) &= \sum_{n=0}^{d}(-1)^{n+1}\frac{C_d^n}{(c+n)^2} \\ &= \sum_{n=1}^{d-1}(-1)^{n+1}\frac{C_{d-1}^n + C_{d-1}^{n-1}}{(c+n)^2} + (-1)^{d+1}\frac{1}{(c+d)^2} - \frac{1}{c^2} \\ &= \sum_{n=1}^{d-1}(-1)^{n+1}\frac{C_{d-1}^n}{(c+n)^2} + (-1)\frac{1}{c^2} \\ &\quad + \sum_{n=1}^{d-1}(-1)^{n+1}\frac{C_{d-1}^{n-1}}{(c+n)^2} + (-1)^{d+1}\frac{1}{(c+d)^2} \\ &= \sum_{n=0}^{d-1}(-1)^{n+1}\frac{C_{d-1}^n}{(c+n)^2} + \sum_{n=0}^{d-1}(-1)^{n+2}\frac{C_{d-1}^n}{(c+1+n)^2} \\ &= f_c(d) - f_{c+1}(d) \\ &= g_c(d) - g_{c+1}(d) \quad \text{(by induction)} \\ &= (-1)\Big(\frac{(c-1)!(d-1)!}{(c+d-1)!} - \frac{c!(d-1)!}{(c+d)!}\Big)\Big(\sum_{n=0}^{d}\frac{1}{c+n}\Big) \\ &= (-1)\frac{(c-1)!d!}{(c+d)!}\Big(\sum_{n=0}^{d}\frac{1}{c+n}\Big) \\ &= g_c(d+1) \end{aligned}$$

Therefore, we have proved $f_c(d) = g_c(d)\ \forall\ c, d \geq 1$, which immediately leads to the proof of this lemma. □

Proof. (Theorem 1)

$$\begin{aligned} KL(\theta_f \parallel \theta_p) &= \int_{-\infty}^{+\infty} p(\theta)\ln\frac{p(\theta)}{q(\theta)}d\theta \\ &= \int_0^1 \frac{1}{\mathcal{B}(c,d)}\theta^{c-1}(1-\theta)^{d-1}\ln\frac{\frac{1}{\mathcal{B}(c,d)}\theta^{c-1}(1-\theta)^{d-1}}{\frac{1}{\mathcal{B}(a,b)}\theta^{a-1}(1-\theta)^{b-1}}d\theta \\ &= \ln\frac{\mathcal{B}(a,b)}{\mathcal{B}(c,d)} + \frac{1}{\mathcal{B}(c,d)}\Big[(c-a)\int_0^1 \theta^{c-1}(1-\theta)^{d-1}\ln\theta d\theta \\ &\quad +(d-b)\int_0^1 \theta^{c-1}(1-\theta)^{d-1}\ln(1-\theta)d\theta\Big] \\ &= \ln\frac{\mathcal{B}(a,b)}{\mathcal{B}(c,d)} + \frac{1}{\mathcal{B}(c,d)}\big[(c-a)\mathcal{B}(c,d)[\Psi(c)-\Psi(c+d)] \\ &\quad +(d-b)\mathcal{B}(c,d)[\Psi(d)+\Psi(c+d)]\big] \quad \text{(Lemma 2)} \\ &= \ln\frac{\mathcal{B}(a,b)}{\mathcal{B}(c,d)} + (c-a)[\Psi(c)-\Psi(c+d)] \\ &\quad +(d-b)[\Psi(d)-\Psi(c+d)] \end{aligned}$$

□