Statistical Debugging using Latent Topic Models [*]

David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu

Computer Sciences Department, University of Wisconsin, Madison WI 53706, USA

Abstract. Statistical debugging uses machine learning to model program failures and help identify root causes of bugs. We approach this task using a novel Delta-Latent-Dirichlet-Allocation model. We model execution traces attributed to failed runs of a program as being generated by two types of latent topics: normal usage topics and bug topics. Execution traces attributed to successful runs of the same program, however, are modeled by usage topics only. Joint modeling of both kinds of traces allows us to identify weak bug topics that would otherwise remain undetected. We perform model inference with collapsed Gibbs sampling. In quantitative evaluations on four real programs, our model produces bug topics highly correlated to the true bugs, as measured by the Rand index. Qualitative evaluation by domain experts suggests that our model outperforms existing statistical methods for bug cause identification, and may help support other software tasks not addressed by earlier models.

1 Introduction

We all depend on buggy software. Computers and computer failures are inescapable features of modern life. As software grows ever more complex and more dynamic, perfectly predicting the (mis)behavior of a software application becomes impossible both in theory and in practice. Therefore, we see increasing interest in *statistical debugging*: the use of statistical machine learning to support debugging. Statistical methods can cope with uncertain and incomplete information while still providing best-effort clues about the causes of software failure. In particular, one can collect examples of successful and failed (e.g., crashed) program runs, then use machine learning techniques to identify those software actions which are strongly associated with program failure. Our goal is *not* to predict whether a run succeeded or failed, but to identify potentially multiple types of bugs in the program.

In contrast with earlier work [1–8] we approach this task using *latent topic models*. These models, such as probabilistic Latent Semantic Analysis [9] and Latent Dirichlet Allocation (LDA [10]), have been successfully applied to model natural language documents [11], images [12] and so on. The contribution of the present work is two-fold:

1. To the best of our knowledge, our work is the first to apply latent topic models to debugging. We employ a novel variant of the LDA model. Each run of a program yields a record of its execution behavior. This record is our document; the words in the document are the events that have been recorded. We describe these records in greater

[*] This research was supported in part by AFOSR Grant FA9550-07-1-0210, NSF Grant CCF-0621487, and NLM Training Grant 5T15LM07359.

detail in section 2. We assume that there are multiple hidden *bug topics*, each with its own multinomial word distribution. The record for each failed run consists partly of words generated from a mixture of the bug topics. The task is to automatically infer the bug topics and mixing weights from multiple runs of the program. We would prefer that bug topics and bug causes had a one-to-one correspondence. This is not a property that our analysis guarantees, but we have found that in practice it is likely.

2. Our latent topic model, Delta Latent Dirichlet Allocation (ΔLDA), can identify weak bug topics from strong interference, while existing latent topic models cannot. In statistical debugging, our primary interest is in the bug topics. For example, a particular bug might trigger a specific segment of code, and produce the corresponding words. However, in a typical run such bug word patterns are overwhelmed by much stronger *usage* word patterns (e.g., code to open a file or to print a page), which are executed more frequently and produce more words. As shown in the literature [8] as well as in our experiments, many standard models are confused by usage patterns and cannot identify bug topics satisfactorily. We explicitly model both bug topics and usage topics on a collection of reports from both failed and successful runs. ΔLDA models successful runs using *only usage topics*, and failed runs with *both usage and bug topics*. Thus, the bug topics are forced to explain the differences between successful runs and failed runs, hence the name ΔLDA.

We review concepts of statistical debugging in section 2, present the ΔLDA model and its collapsed Gibbs sampling inference procedure in section 3, and demonstrate its effectiveness for debugging with both a synthetic example and four real programs, i.e., `exif`, `grep`, `gzip`, and `moss`, in section 4. For the task of helping humans identify root causes of bugs, our ΔLDA model performs as well or better than the best previously-proposed statistical methods. Furthermore, it supports related debugging tasks not contemplated by prior work. These benefits are all built upon a single integrated model with a coherent interpretation in both machine-learning and software-engineering terms.

2 Cooperative Bug Isolation

The Cooperative Bug Isolation Project (CBI) is an ongoing effort to enlist large user communities to isolate and ultimately repair the causes of software bugs [13]. Statistical debugging is a critical component of CBI as it allows us to cope with unreliable and incomplete information about failures in deployed software systems. In this section we briefly review the CBI approach and infrastructure to show how it maps software behavior into a document-and-word model suitable for latent topic analysis.

The data for CBI analysis consists of reports generated by instrumented versions of software applications. The code inserted by the CBI instrumentor passively logs many program-internal events of potential interest to bug-hunting software engineers while the software application is being executed. Interesting events may include the direction taken when a branch (`if` statement) is executed, whether a function call returns a negative, zero, or positive result, the presence of unusual floating-point values, and so forth. Via the instrumentation, each run of a program generates a sequence of recorded events. This sequence is the “document”; the recorded events are the “word tokens”. The set of all possible events that can be recorded by the instrumentation code corresponds to the set of "word types".

Instrumentation code is distributed throughout the source code of a program. Even a medium-sized program can have hundreds of thousands of instrumentation points (word types); a single event (word token) may occur millions of times during a single run. For reasons of performance, scalability, and user privacy, we cannot record every event. Instead, events are sparsely sampled during each run. A typical sampling rate is 1⁄100, meaning each event has only a 1⁄100 chance of being observed and recorded each time it occurs. More sophisticated instrumentation can adapt the sampling rate to the expected number of occurrences of a given event, sampling rare events at a higher rate and common events at a lower rate, and thereby increasing the probability that a rare event will be recorded if it occurs. A second practical measure is to discard all event ordering information, and instead report the number of times an event was recorded. A single run, then, results in a single fixed-length vector of event counts, called a *feedback report*. The data in any single feedback report is an incomplete but unbiased random sample of the behavior during that run. In machine-learning terms, a feedback report is a "bag of words" representation of the document generated by a run.

Feedback reports are collected centrally for aggregation and analysis. Reports may come from real users participating in the ongoing CBI public deployment [14] or may be produced in-house with fixed or randomly-generated test suites. Each feedback report carries one additional piece of information: an *outcome flag* recording whether this run succeeded or failed. In the simplest case, all fatal software crashes might be considered "failures" and all non-crashing runs considered "successes." More sophisticated flagging strategies such as comparing program output against that of a known-good reference implementation may also be used.

For any program of non-trivial complexity, we must further assume that there are an unknown number of latent bugs. Because instrumentation is so broad, we must assume that the vast majority of program events are not directly connected to any given bug. Thus, the bug "signal" is both noisy due to sparse sampling and weak relative to the majority non-buggy behavior of the program.

The statistical debugging challenge, then, is as follows. **Given** a large collection of feedback reports of a program, where each report is flagged according to whether the run succeeded or failed, and where there may be a number of bugs, i.e., causes for failure, **distinguish** among these causes of failure, **identify** events that contributed to a failure and are connected with its underlying cause, and **use** this information to help support the debugging process in particular and software understanding more broadly.

3 The ΔLDA Model

Standard LDA [10] models a single document collection. For example, when applied to a collection of failed runs only, standard LDA is likely to recover stronger usage patterns rather than generally weaker bug patterns. In contrast, ΔLDA models a mixed collection of successful and failed runs. We reserve extra bug topics for failed runs in order to capture the weaker bug patterns. By explicitly modeling successful vs. failed

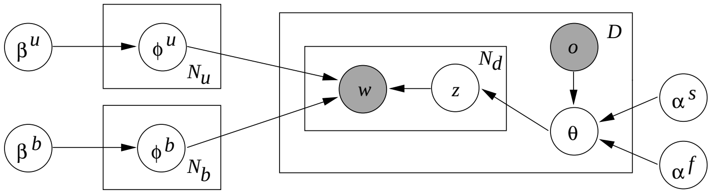


Fig. 1: The ΔLDA model

runs, and usage vs. bug topics, ΔLDA is able to recover the weak bug topics more clearly. The ΔLDA model (Figure 1) has the following major components:

(i) There are N_u usage topics ϕ^u, and N_b bug topics ϕ^b. These are sampled from two Dirichlet distributions: $\phi^u \sim \text{Dir}(\beta^u)$, $\phi^b \sim \text{Dir}(\beta^b)$. We distinguish β^u and β^b (instead of a single β) to facilitate the incorporation of certain types of domain knowledge. For example, if we believe that some parts of the software are more error-prone (e.g., less tested) than others, then the bug topics may focus more on the corresponding words.

(ii) There are a total of D documents. Each document has an observed outcome flag $o \in \{s, f\}$ for successful and failed run, respectively. These D documents constitute the mixed collection of successful and failed runs.

(iii) Each document is generated as a "bag of words" by a mixture of the $N_u + N_b$ topics. The mixing weight θ is sampled from one of two Dirichlet distributions α^s or α^f, depending on the outcome flag o: $\theta \sim \text{Dir}(\alpha^o)$. In the simplest case, the elements in α^s that correspond to bug topics are set to zero, ensuring that any successful run will not use any bug topic[1]. By contrast, all the elements of α^f are greater than zero, allowing failed runs to use both usage and bug topics.

(iv) The rest of the model is identical to LDA: for each of the N_d word positions in the document, one samples a topic index $z \sim \text{Multi}(\theta)$, $z \in \{1, \ldots, N_u + N_b\}$, and produces a word $w \sim \text{Multi}(\phi_z)$.

The ΔLDA model thus specifies the conditional probability $p(\mathbf{w}|\mathbf{o}, \beta^u, \beta^b, \alpha^s, \alpha^f)$, where we use bold face to denote sequences of variables. Omitting hyperparameters for notational simplicity, this can be computed as $p(\mathbf{w}|\mathbf{o}) = \sum_{\mathbf{z}} p(\mathbf{w}|\mathbf{z}) p(\mathbf{z}|\mathbf{o})$, where

$$p(\mathbf{w}|\mathbf{z}) = \prod_{i}^{N_u+N_b} \int p(\phi_i|\beta^u, \beta^b) \prod_{j}^{W} {\phi_{ij}}^{n^i_j} \, \mathrm{d}\phi_i \tag{1}$$

$$p(\mathbf{z}|\mathbf{o}) = \prod_{d}^{D} \int p(\theta_d|o_d, \alpha^s, \alpha^f) \prod_{i}^{N_u+N_b} {\theta_{di}}^{n^d_i} \, \mathrm{d}\theta_d \; . \tag{2}$$

Here W is the vocabulary size, n^i_j is the number of times word-type j is assigned to topic i, and n^d_i is the number of times topic i occurs in document d. Also, ϕ_{ij} is the probability of word j being generated by topic i and θ_{di} is the probability of using topic i in document d.

[1] It is straightforward to allow small but non-zero bug topic weights for successful runs. This is useful if we believe some runs were affected by bugs but did not fail.

3.1 Inference

We are interested in the hidden variables $\mathbf{z}, \theta, \phi$. We can draw $\mathbf{z}$ samples from the posterior $p(\mathbf{z}|\mathbf{w}, \mathbf{o})$ using Markov Chain Monte Carlo (MCMC). In particular, we use collapsed Gibbs sampling [11], drawing from $p(z_k = i|\mathbf{z}_{-k}, \mathbf{w}, \mathbf{o})$ for each site k in sequence. This inference procedure is linear in the number of samples taken, the total number of topics used, and the size of the corpus. Since

$$p(z_k = i|\mathbf{z}_{-k}, \mathbf{w}, \mathbf{o}) = \frac{p(z_k = i, \mathbf{z}_{-k}, \mathbf{w}|\mathbf{o})}{\sum_{i'} p(z_k = i', \mathbf{z}_{-k}, \mathbf{w}|\mathbf{o})} , \tag{3}$$

the site conditionals can be computed from the joint $p(\mathbf{z}, \mathbf{w}|\mathbf{o}) = p(\mathbf{z}|\mathbf{o})p(\mathbf{w}|\mathbf{z})$, as given in (1) and (2). The Dirichlet priors can then be integrated out (“collapsed”) in (1) and (2), resulting in the following multivariate Pólya distributions:

$$p(\mathbf{w}|\mathbf{z}) = \prod_{i}^{N_u+N_b} \left[\frac{\Gamma(\sum_{j'}^{W} \beta^i_{j'})}{\Gamma(\sum_{j'}^{W} \beta^i_{j'} + n^i_*)} \prod_{j}^{W} \frac{\Gamma(n^i_j + \beta^i_j)}{\Gamma(\beta^i_j)} \right] \tag{4}$$

$$p(\mathbf{z}|\mathbf{o}) = \prod_{d}^{D} \left[\frac{\Gamma(\sum_{i'}^{N_u+N_b} \alpha^{o_d}_{i'})}{\Gamma(\sum_{i'}^{N_u+N_b} \alpha^{o_d}_{i'} + n^d_*)} \prod_{i}^{N_u+N_b} \frac{\Gamma(n^d_i + \alpha^{o_d}_i)}{\Gamma(\alpha^{o_d}_i)} \right] . \tag{5}$$

Here n^i_* is the count of all words assigned to topic i, and n^d_* is the count of all words contained in document d. β^i_j is the hyperparameter associated with word j in topic i, where β^i is β^u if i is a usage topic and β^b if it is a bug topic. $\alpha^{o_d}_i$ is the hyperparameter associated with topic i for outcome flag value o_d. Rearranging (4) and (5) yields

$$p(z_k = i|\mathbf{z}_{-k}, \mathbf{w}, \mathbf{o}) \propto \left(\frac{n^i_{-k,j_k} + \beta^i_{j_k}}{n^i_{-k,*} + \sum_{j'}^{W} \beta^i_{j'}} \right) \left(\frac{n^{d_k}_{-k,i} + \alpha^{o_k}_i}{n^{d_k}_{-k,*} + \sum_{i'}^{N_u+N_b} \alpha^{o_k}_{i'}} \right) . \tag{6}$$

In this equation, all “n_{-k}” are counts excluding the word or topic assignment at position k. Also, j_k is the word at position k, d_k is the document containing position k, and o_k is the outcome flag associated with d_k. This equation allows us to perform collapsed Gibbs sampling efficiently using easily obtainable count values. Note that for topics i such that $\alpha^{o_k}_i = 0$, the count $n^d_{-k,i}$ is also 0, meaning that topic i will never be assigned to this word.

After the MCMC chain mixes, we can use a single sample from the posterior $p(\mathbf{z}|\mathbf{w}, \mathbf{o})$ to estimate ϕ_i, the multinomial over words for topic i, and θ_d, the topic mixture weights for document d:

$$\hat{\phi}_{ij} = \frac{n^i_j + \beta^i_j}{n^i_* + \sum_{j'}^{W} \beta^i_{j'}} \tag{7}$$

$$\hat{\theta}_{di} = \frac{n^d_i + \alpha^{o_d}_i}{n^d_* + \sum_{i'}^{N_u+N_b} \alpha^{o_d}_{i'}} . \tag{8}$$

We use domain expert knowledge to set the hyperparameters α^s, α^f, β^u, β^b, as well as the number of usage and bug topics N_u, N_b. Hyperparameter values used are not specially fitted to our data and should perform well in a variety of situations. Alternatively, these values could be estimated from the data using Bayesian model evidence maximization. This involves finding the values which maximizes the evidence, $p(\mathbf{w}|\mathbf{o})$. In

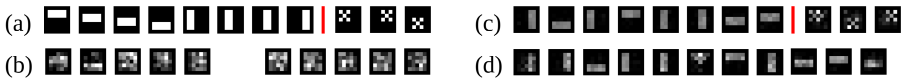


Fig. 2: A toy example showing *Δ*LDA's ability to recover weak bug topics. (a) Truth: 8 usage topics and 3 bug topics; (b) Example success (left) and failure (right) documents; (c) *Δ*LDA successfully recovers the usage and bug topics; (d) Standard LDA cannot recover or identify bug topics.

particular, the Gibbs sampling technique employed by our model allows the convenient estimation of the evidence by importance sampling, using $\mathbf{z}$ samples drawn from our MCMC chain [15].

4 Experiments

4.1 A Toy Example

We first use a toy dataset to demonstrate *Δ*LDA's ability to identify bug topics. The vocabulary consists of 25 pixels in a 5-by-5 grid. We use 8 usage topics and 3 bug topics as in Figure 2(a). Each of the 8 usage topics corresponds to a uniform distribution over a 2-pixel wide horizontal or vertical bar. Each of the 3 bug topics corresponds to a uniform distribution over the pixels in a small "x". In all diagrams, each image also has a 1-pixel black frame for visibility, which does not correspond to any vocabulary word. We generate 2000 documents of length 100 with these topics according to the procedure described in Section 3. Half of the documents are "successful runs" and the other half "failed runs." For the topic mixture hyperparameters, we use $\alpha^s = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]$ and $\alpha^f = [1, 1, 1, 1, 1, 1, 1, 1, 0.1, 0.1, 0.1]$. This means that the bug topics are never present in the $o_d = s$ (successful) documents, and tend to be only weakly present in the $o_d = f$ (failed) documents. Some example documents from this generated corpus are shown in Figure 2(b).

We then run *Δ*LDA and standard LDA [11, using code at `http://psiexp.ss.uci.edu/research/programs_data/toolbox.htm`] on the toy dataset. In order to give standard LDA the best chance of identifying the bug topics, it is run on $o_d = f$ documents only[2], using 11 topics. *Δ*LDA is run on all documents, using 8 usage topics and 3 bug topics. *Δ*LDA is supplied with the true α vectors used to generate the data, but the standard LDA implementation used in this experiment only allows a symmetric α hyperparameter (where all values in the α vector have the same value). Therefore we supply the standard LDA model with the symmetric hyperparameter $\alpha = 1$. Further experiments (not shown here) using a different implementation of standard LDA and the true α^f vector achieve similar results. Both models use the same symmetric hyperparameter $\beta = 1$ (which is not actually used to generate the data because the topics are fixed). Both MCMC chains are run for 2000 full samples, after which ϕ and θ are estimated from the final sample as described above.

[2] Additional experiments (not shown here) validate the intuition that the inclusion of $o_d = s$ documents does not improve the recovery of bug topics with standard LDA.

Table 1: General information about test programs.

			Runs			Topics	
Program	Lines of Code	Bugs	Successful	Failing	Word Types	Usage	Bug
`exif` [16]	10,611	2	352	30	20	7	2
`grep` [17, 18]	15,721	2	609	200	2,071	5	2
`gzip` [17, 18]	8,960	2	29	186	3,929	5	2
`moss` [19]	35,223	8	1,727	1,228	1,982	14	8

The estimated topics for *Δ*LDA are shown in Figure 2(c), and the estimated topics for standard LDA are shown in Figure 2(d). *Δ*LDA is able to recover the true underlying usage and bug topics quite cleanly. On the other hand, standard LDA is unable to separate and identify bug topics, either mixing them with usage topics or simply duplicating usage topics. The toy example clearly shows the superiority of *Δ*LDA over standard LDA in this setting.

4.2 Real Programs

While *Δ*LDA performs well on our toy example, real programs are orders of magnitude more complex. We have applied *Δ*LDA to CBI feedback reports from four buggy C programs, details of which appear in Table 1. Bugs are naturally-occurring (`exif`), hand-seeded (`grep`, `gzip`), or both (`moss`). All four programs are in use by real users or are directly derived from real-world code. Test inputs are randomly-generated among reasonable inputs for each program, and “failure” is defined as crashing or producing output different from a known-correct reference implementation. Hand-coded “bug oracles” provide ground truth as to which bugs were actually triggered in any given run. For our experiments on `grep` and `gzip` we used test suites supplied by the SIR repository developers [18]. For `exif` and `moss` tests were generated using randomly selected command line flags and inputs. Feedback data is non-uniformly sampled during program execution as in prior work [5].

For these experiments, all hyperparameters used are symmetric, and the same hyperparameter settings are used for all programs. We set $\beta^u = \beta^b = 0.1$, and nonzero entries of $\alpha^s = \alpha^f = 0.5$ to encourage sparsity. The number of topics for each program are chosen according to domain expert advice. The number of bug topics N_b for each program is set equal to the number of distinct bugs known to be manifested in our dataset, with the goal of characterizing each bug with a single topic. The number of usage topics N_u for each program is chosen to approximately correspond to the number of different program use cases. For example, `exif` uses seven different mutually-exclusive command line flags, each of which corresponds to a different program operation. Therefore, it is natural to model the program usage patterns with seven different usage topics. Table 1 gives the number of usage and bug topics used for each program. For all programs, the Gibbs chain is run for 2000 iterations before estimating ϕ and θ. For the largest dataset, `moss`, the inference step took less than one hour to run on a desktop workstation.

Where possible, we compare *Δ*LDA results with corresponding measures from two earlier statistical debugging techniques. *PLDI05* refers to earlier work by Liblit et al. [5] that uses an iterative process of selecting and eliminating top-ranked predicates until all failures are explained. The approach bears some resemblance to likelihood ratio testing and biased minimum-set cover problems, but is somewhat *ad hoc* and highly specialized for debugging. *ICML06* refers to earlier work by Zheng et al. [8] that takes inspiration from bi-clustering algorithms. This approach uses graphical models to estimate complete (non-sampled) counts, then applies an iterative collective voting scheme followed by a simple clustering pass to identify and report likely bug causes.

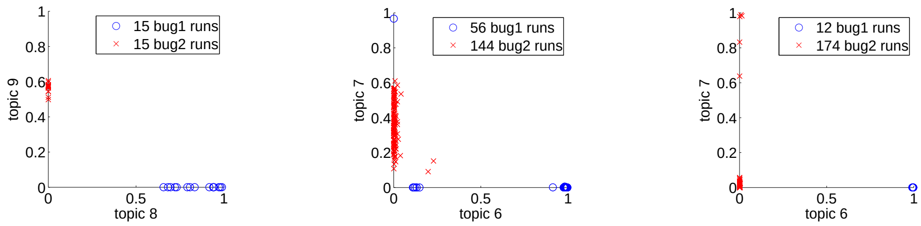
Bug Topic Analysis on θ. We show that *Δ*LDA is capable of recovering bug topics that correlates well with the underlying software bugs. Note that each failed run's N_b bug topic elements in θ, which we call θ^b, can be viewed as a low-dimensional bug representation of failed runs. We plot the failed runs in this θ^b space for the programs in Figure 3(a-d), where we use different symbols to mark the failed runs by their ground truth bug types. In the case of moss, we project the 8-dimensional θ^b space down to 3 dimensions for visualization using Principal Component Analysis. The plots show that actual bug types tend to cluster *along the axes* of θ^b, which means that often a *Δ*LDA bug topic maps to a unique actual bug type. Multi-bug runs exhibit multiple high-weight θ^b components, which is consistent with this mapping between bug topics and actual bugs. This observation could be used to focus debugging efforts on single-bug runs.

Figure 3(e) compares the quality of clusterings given by *Δ*LDA to those of the other analyses. For each analysis we compute the Rand index [20] of a clustering based on that analysis with respect to the ground truth (determined by our oracle). A Rand index of 1 indicates that the clustering is identical to the ground truth; lower indices indicate worse agreement. For *Δ*LDA, we assign a failed run to cluster i if its bug topic i has the largest θ_i among all bug topics. Other clustering methods are possible and may produce better clusters. No method dominates, but *Δ*LDA consistently performs well.

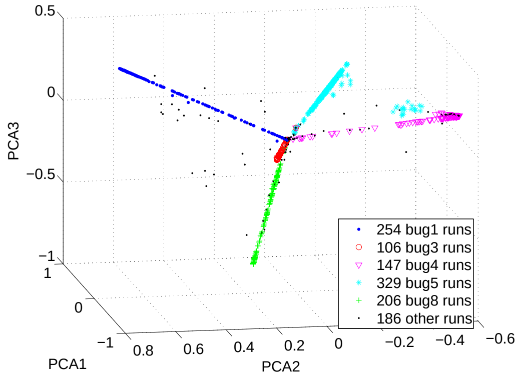
Bug Topic Analysis on ϕ. We now discuss how to extract ranked lists of suspect words (potentially buggy program behaviors) for each bug topic. We qualitatively evaluate the usefulness of these lists in finding root causes of bugs, both for *Δ*LDA and for PLDI05 and ICML06. Overall, we find that *Δ*LDA and PLDI05 perform equally well, with *Δ*LDA resting on a stronger mathematical foundation and potentially supporting a wider variety of other important tasks.

The parameter ϕ itself specifies $p(w|z)$. But a word w may have a large $p(w|z)$ simply because it is a frequent word in all topics. We instead examine $p(z|w)$ which is easily obtained from ϕ using Bayes rule. Furthermore, we define a confidence score $S_{ij} = \min_{k \neq i} p(z = i|w = j) - p(z = k|w = j)$. For each topic, $z = i$, we present words ranked by their score, S_{ij}. If S_{ij} is less than zero, indicating that word j is more predictive of some other topic, we do not present the word at all. On the other hand, if S_{ij} is high, then we consider that word j is a suspect word and a likely cause of the bug explained by topic i.

In lieu of formal human-subject studies, which are outside the scope of this paper, we informally assess the expected usefulness of these suspect-word lists to a bug-



(a) `exif` bug topics (b) `grep` bug topics (c) `gzip` bug topics



(d) `moss` bug topics (with PCA)

	ΔLDA	ICML06	PLDI05
`exif`	1.00	0.88	1.00
`grep`	0.97	0.71	0.77
`gzip`	0.89	1.00	1.00
`moss`	0.93	0.93	0.96

(e) Rand index

Fig. 3: ΔLDA recovers bug topics that highly correlate with actual software bugs

hunting programmer. We compare ΔLDA results with analogous lists built using the PLDI05 and ICML06 algorithms mentioned earlier.

For `exif`, the two ΔLDA bug topics cleanly separate the two bugs. Each topic's list of suspect words is short (4 and 6 items) but relevant. For one of the bugs, this list includes a clear "smoking gun" that immediately reveals the root cause; the other bug topic includes less direct secondary effects of the root problem. PLDI05 performs well for both `exif` bugs. However, while ΔLDA's word list is naturally restricted to words for which $S_{ij} > 0$, PLDI05 has no intuitive cut-off point. Thus, PLDI05's lists include all words under analysis (19 for `exif`) and risk overwhelming programmers with irrelevant information. In our subjective experience, if the first five or ten items in a "suspect program behaviors" list are not immediately understandable, the programmer is unlikely to search further. ICML06 struggles with this as well. If clustering is not used, ICML06 reports all 19 words without good separation into bug-specific groups. If clustering is used, ICML06 offers a short 3-item list that describes one bug three times and omits the other bug entirely.

`moss` results vary in quality from bug to bug. Overall, we find that most ΔLDA bug topics correspond directly to individual `moss` bugs, and that highly-suspect words for each of these topics often identify either primary "smoking gun" causes for failure or else closely related secondary effects of the initial misbehavior. PLDI05 performs better than *Δ*LDA for some bugs and worse for others, with each analysis identifying at least one smoking gun that the other misses. ICML06 with clustering produces identifies the smoking gun for one bug that both *Δ*LDA and PLDI05 miss. However, ICML06 reports thirty clusters while there are only eight actual `moss` bugs: several bugs are split among multiple clusters and therefore presented redundantly.

`grep` and `gzip` results are equivocal. ICML06 identifies an informative precondition for one `grep` bug, though not the smoking gun. Otherwise, all three algorithms identify words that are strongly associated with bugs but which do not immediately reveal any bugs' root causes. These algorithms do not truly model causality, and therefore it is not surprising that root causes may be difficult to recover. Furthermore, in some cases, no smoking guns were actually among the words instrumented and considered by the models. We feel that all three models perform as well as can be reasonably expected given the less-than-ideal raw data.

Overall, we find that the PLDI05 and *Δ*LDA approaches perform roughly equally well in guiding humans to the root causes of bugs. However, PLDI05 is highly specialized and somewhat difficult to reason about formally. For example, whereas *Δ*LDA ranks words using conditional probabilities, PLDI05 computes multi-factor harmonic mean scores that, while about as effective, have no simple interpretation either in machine learning terms or as quantitative measures of expected program behavior. *Δ*LDA has, we assert, a stronger mathematical foundation and potentially broader applicability to problems in other domains (see section 5).

Furthermore, even within the domain of statistical debugging, components of an *Δ*LDA model can be used to support other important software engineering tasks not contemplated by earlier approaches. Suppose, for example, that one's task is to fix the bug associated with a particular bug topic, and that a repeatable test suite is available. In that case, one would prefer to investigate runs where the weight for that bug topic is very high compared to the weight for all other bug topics, as those runs would be likely to be the most pure embodiments of the bug. For another example, prior work has shown how to automatically construct extended paths through multiple suspect program points [21]; *Δ*LDA offers a model whereby the aggregate scores along such paths can be given a sensible probabilistic interpretation. While we have not yet explored these alternate uses in detail, they hint at the power of a statistical debugging approach that is both well-founded in theory and highly effective in practice.

Usage Topic Analysis Information gleaned from usage topics might support a variety of software engineering tasks. To characterize a usage topic, we examine the words that have the highest probability conditioned on that topic. Each word is associated with the source code immediately adjacent to its instrumentation point. We find that in many cases usage topics correspond to distinct usage modes of the program.

We describe `gzip` in detail, since the DEFLATE algorithm which it implements is in the public domain and likely to be familiar to many in the machine learning community. Recall that the DEFLATE algorithm consists of two steps, duplicate string elimination and bit reduction using Huffman coding.

For each usage topic there is a small number of highly probable words and a much larger number that are significantly less probable. The most probable word by far in topic 1 is associated with an inner loop in longest_match(), the underlying procedure in the duplicate string elimination step of the algorithm. We infer that topic 1 is highly associated with this step. We expect runs with a high $p(z = 1|d)$ value to use the algorithm which finds the most redundant strings and does the best compression at the expense of running more slowly; this is the case. There are about twenty highly probable words in topic 2; all are associated with command line handling or exit clean-up code. In runs where $p(z = 2|d)$ is relatively high no compression occurred; instead, for example, a help message or version message was requested. Of the twenty most probable words in topic 3, several are associated with longest_match(), several with compress_block(), and several with deflate_fast(). We infer that this usage topic is associated with the fast deflation algorithm which does only very simple duplicate string elimination. In the runs where $p(z = 3|d)$ is highest, gzip was invoked with a flag explicitly calling for the fast algorithm. The modes associated with topics 4 and 5 are less pronounced. Topic 4 seems to capture output activity, as it includes a highly probable word in in write_buf() as well as a few highly probable words associated with the duplicate string elimination algorithm. Topic 5 seems to capture the bit reduction mode, as words in updcrc(), a utility function used for shifting bits, are by far the most probable.

5 Conclusions and Discussion

Software continues to be released with bugs, and end users suffer the consequences. However, statistical models applied to instrumented feedback data can help programmers repair problems. ΔLDA shows promise as a statistical model with both strong empirical results and a sound mathematical foundation. Some future directions have been suggested earlier, such as incorporating domain knowledge into the Dirichlet hyperparameters or automatically identifying the number of bugs. Another direction is to endow ΔLDA with more complex topic structure similar to Hierarchical LDA [22], which arranges topics in a tree. However, Hierarchical LDA provides no mechanism for document-level control (the outcome flag) on topic availability. Other modifications to the model could exploit some of the interesting structure inherent in this problem domain, such as the static program graph.

Note that ΔLDA need not be restricted to statistical debugging. For example, ΔLDA may be applied to text sentiment analysis [23] to distinguish "subjective sentiment topics" (e.g., positive or negative opinions, the equivalent of bug topics) from much stronger "objective content topics" (in the movie domain these are movie plots, actor biographies etc., the equivalent of usage topics). For the movie domain, the mixed document collection may consist of user-posted movie reviews (which contain both sentiment and content topics), and formal movie summaries (which contain mostly objective contents).

References

1. Arumuga Nainar, P., Chen, T., Rosin, J., Liblit, B.: Statistical debugging using compound boolean predicates. In Elbaum, S., ed.: International Symposium on Software Testing and Analysis, London, United Kingdom (July 9–12 2007)

2. Dickinson, W., Leon, D., Podgurski, A.: Finding failures by cluster analysis of execution profiles. In: Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01), IEEE Computer Society (2001) 339–348
3. Hangal, S., Lam, M.S.: Tracking down software bugs using automatic anomaly detection. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering, New York, NY, USA, ACM Press (2002) 291–301
4. Jones, J.A., Harrold, M.J.: Empirical evaluation of the Tarantula automatic fault-localization technique. In: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, New York, NY, USA, ACM Press (2005) 273–282
5. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, Illinois (June 12–15 2005)
6. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: SOBER: statistical model-based bug localization. In Wermelinger, M., Gall, H., eds.: ESEC/SIGSOFT FSE, ACM (2005) 286–295
7. Zheng, A.X., Jordan, M.I., Liblit, B., Aiken, A.: Statistical debugging of sampled programs. In Thrun, S., Saul, L., Schölkopf, B., eds.: NIPS 16. MIT Press, Cambridge, MA (2004)
8. Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A.: Statistical debugging: Simultaneous identification of multiple bugs. In: ICML. (2006)
9. Hofmann, T.: Probabilistic latent semantic analysis. In: Proc. of Uncertainty in Artificial Intelligence, UAI'99, Stockholm (1999)
10. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. Journal of Machine Learning Research **3** (2003) 993–1022
11. Griffiths, T., Steyvers, M.: Finding scientific topics. Proceedings of the National Academy of Sciences **101**(suppl. 1) (2004) 5228–5235
12. Fei-Fei, L., Perona, P.: A Bayesian hierarchical model for learning natural scene categories. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2005)
13. Liblit, B.: Cooperative Bug Isolation: Winning Thesis of the 2005 ACM Doctoral Dissertation Competition. Volume 4440 of Lecture Notes in Computer Science. Springer (2007)
14. Liblit, B.: The Cooperative Bug Isolation Project. `http://www.cs.wisc.edu/cbi/`
15. Kass, R., Raftery, A.: Bayes factors. Journal of the American Statistical Association **90** (1995) 773–795
16. EXIF Tag Parsing Library. `http://libexif.sf.net/`
17. H. Do, S. Elbaum, G.R.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal **10**(4) (2005) 405–435
18. Rothermel, G., Elbaum, S., Kinneer, A., Do, H.: Software-artifact intrastructure repository. `http://sir.unl.edu/portal/` (September 2006)
19. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In ACM, ed.: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003, New York, NY 10036, USA, ACM Press (2003) 76–85
20. Rand, W.M.: Objective criteria for the evaluation of clustering methods. Journal of the American Statistical Association **66** (1971) 846–850
21. Lal, A., Lim, J., Polishchuk, M., Liblit, B.: Path optimization in programs and its application to debugging. In Sestoft, P., ed.: 15th European Symposium on Programming, Vienna, Austria, Springer (March 2006) 246–263
22. Blei, D.M., Griffiths, T.L., Jordan, M.I., Tenenbaum, J.B.: Hierarchical topic models and the nested Chinese restaurant process. In: NIPS 16. (2003)
23. Pang, B., Lee, L.: A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In: Proceedings of the Association for Computational Linguistics. (2004) 271–278