Learning Tractable Probabilistic Models for Fault Localization

Aniruddh Nath* and **Pedro Domingos**
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195, U.S.A.
{*nath, pedrod*}*@cs.washington.edu*

Abstract

In recent years, several probabilistic techniques have been applied to various debugging problems. However, most existing probabilistic debugging systems use relatively simple statistical models, and fail to generalize across multiple programs. In this work, we propose *Tractable Fault Localization Models* (TFLMs) that can be learned from data, and probabilistically infer the location of the bug. While most previous statistical debugging methods generalize over many executions of a single program, TFLMs are trained on a corpus of previously seen buggy programs, and learn to identify recurring patterns of bugs. Widely-used fault localization techniques such as TARANTULA evaluate the suspiciousness of each line in isolation; in contrast, a TFLM defines a joint probability distribution over buggy indicator variables for each line. Joint distributions with rich dependency structure are often computationally intractable; TFLMs avoid this by exploiting recent developments in tractable probabilistic models (specifically, Relational SPNs). Further, TFLMs can incorporate additional sources of information, including coverage-based features such as TARANTULA. We evaluate the fault localization performance of TFLMs that include TARANTULA scores as features in the probabilistic model. Our study shows that the learned TFLMs isolate bugs more effectively than previous statistical methods or using TARANTULA directly.

Introduction

According to a 2002 NIST study (RTI International 2002), software bugs cost the US economy an estimated $59.5 billion per year. While some of these costs are unavoidable, the report claimed that an estimated $22.2 billion could be saved with more effective tools for the identification and removal of software errors. Several other sources estimate that over 50% of software development costs are spent on debugging and testing (Hailpern and Santhanam 2002).

The need for better debugging tools has long been recognized. The goal of automating various debugging tasks has motivated a large body of research in the software engineering community. However, this line of work has only recently begun to take advantage of recent advances in probabilistic models, and their inference and learning algorithms.

In this work, we apply state-of-the-art probabilistic methods to the problem of fault localization. We propose *Tractable Fault Localization Models* (TFLMs) that can be learned from a corpus of known buggy programs (with the bug locations annotated). The trained model can then be used to infer the probable locations of buggy lines in a previously unseen program. Conceptually, a TFLM is a probability distribution over programs in a given language, modeled jointly with any attributes of interest (such as bug location indicator variables, or diagnostic features). Conditioned on a specific program, a TFLM defines a joint probability distribution over the attributes.

The key advantage of probabilistic models is their ability to learn from experience. Many software faults are instances of a few common error patterns, such as off-by-one errors and use of uninitialized values (Brun and Ernst 2004). Human debuggers improve with experience as they encounter more of these common fault patterns, and learn to recognize them in new programs. Automated debugging systems should be able to do the same. Another advantage of probabilistic models is that they allow multiple sources of information to be combined in a principled manner. The relative contribution of each feature determined by its predictive value in the training corpus, rather than by a human expert. A TFLM can incorporate as features the outputs of other fault localization systems, such as the TARANTULA hue (Jones, Harrold, and Stasko 2002) of each line.

In recent years, there has been renewed interest in learning rich, tractable models, on which exact probabilistic inference can be performed in polynomial time (e.g. Sum-Product Networks; Poon and Domingos 2011). TFLMs build on Relational Sum-Product Networks (Nath and Domingos 2015) to enable exact inference in space and time linear in the size of the program. We empirically compare TFLMs to the widely-used TARANTULA fault localization method, as well as the Statistical Bug Isolation (SBI) system, on four mid-sized C programs. TFLMs outperform the other systems on three of the four test subjects.

Background

Coverage-based Fault Localization

Coverage-based debugging methods isolate the bug's location by analyzing the program's coverage spectrum on a set

*Now at Google, Inc.

of test inputs. These approaches take the following as input:

1. a set of unit tests;
2. a record of whether or not the program passed each test;
3. program traces, indicating which components (usually lines) of the program were executed when running each unit test.

Using this information, these methods produce a suspiciousness score for each component in the program. The most well-known method in this class is the TARANTULA system (Jones, Harrold, and Stasko 2002), which uses the following scoring function:

$$S_{Tarantula}(s) = \frac{\frac{Failed(s)}{TotalFailed}}{\frac{Passed(s)}{TotalPassed} + \frac{Failed(s)}{TotalFailed}}$$

Here, $Passed(s)$ and $Failed(s)$ are respectively the number of passing and failing test cases that include statement s, and $TotalPassed$ and $TotalFailed$ are the number of passing and failing test cases respectively. In an empirical evaluation (Jones and Harrold 2005), TARANTULA was shown to outperform previous methods such as cause transitions (Cleve and Zeller 2005), set union, set intersection and nearest neighbor (Renieris and Reiss 2003), making it the state of the art in fault localization at the time. Since the publication of that experiment, a few other scoring functions have been shown to outperform TARANTULA under certain conditions (Abreu et al. 2009). Nonetheless, TARANTULA remains the most well-known method in this class.

Probabilistic Debugging Methods

Per-Program Learning Several approaches to fault localization make use of statistical and probabilistic methods. Liblit et al. proposed several influential statistical debugging methods. Their initial approach (Liblit et al. 2003) used ℓ_1-regularized logistic regression to predict non-deterministic program failures. The instances are runs of a program, the features are instrumented program predicates, and the models are trained to predict a binary 'failure' variable. The learned weights of the features indicate which predicates are the most predictive of failure. In later work (Liblit et al. 2005), they use a likelihood ratio hypothesis test to determine which *predicates* (e.g. branches, sign of return value) in an instrumented program are predictive of program failure. Zhang et al. (2011) evaluate several other hypothesis testing methods in a similar setting.

The SOBER system (Liu et al. 2005; 2006) improves on Liblit et al.'s 2005 approach by taking into account the fact that a program predicate can be evaluated multiple times in a single test case. They learn conditional distributions over the probability of a predicate evaluating to `true`, conditioned on the success or failure of the test case. When these conditional distributions differ (according a statistical hypothesis test), the predicate is considered to be 'relevant' to the bug. The HOLMES system (Chilimbi et al. 2009) extends Liblit et al.'s approach along another direction, analyzing path profiles instead of instrumented predicates.

Wong et al. (2008; 2012) use a crosstab-based statistical analysis to quantify the dependence between statement coverage and program failure. Their approach can be seen as a hybrid between the Liblit-style statistical analysis and TARANTULA-style spectrum-based analysis. Wong et al. (2009; 2012) also proposed two neural network-based fault localization techniques trained on program traces. Ascari et al. (2009) investigate the use of SVMs in a similar setting.

Many of the methods described above operate under the assumption that the program contains exactly one bug. Some of these techniques have nevertheless been evaluated on programs with multiple faults, using an iterative process where the bugs are isolated one by one. Briand et al. (2007) explicitly extend TARANTULA to the multiple-bug case, by learning a decision tree to partition failing test cases. Each partition is assumed to model a different bug. Statements are ranked by suspiciousness using a TARANTULA-like scoring function, with the scores computed separately for each partition. Other clustering methods have also been applied to test cases; for example, Andrzejewski et al. (2007) use a form of LDA to discover latent 'bug topics'.

Generalizing Across Programs The key limitation of the statistical and machine learning-based approaches discussed above is that they only generalize over many executions of a single program. Ideally, a machine learning-based debugging system should be able to generalize over multiple programs (or, at least, multiple sequential versions of a program). As discussed above, many software defects are instances of frequently occurring fault patterns; in principle, a machine learning model can be trained to recognize these patterns and use them to more effectively localize faults in new programs.

This line of reasoning has received relatively little attention in the automated debugging literature. The most prominent approach is the Fault Invariant Classifier (FIC) of Brun and Ernst (2004). FIC is not a fault localization algorithm in the sense of TARANTULA and the other approaches discussed above. Instead of localizing the error to a particular line, FIC outputs *fault-revealing properties* that can guide a human debugger to the underlying error. These properties can be computed using static or dynamic program analysis; FIC uses the Daikon dynamic invariant detector (Ernst et al. 2001). At training time, properties are computed for pairs of buggy and fixed programs; properties that occur in the buggy programs but not the fixed programs are labeled as 'fault-revealing'. The properties are converted into program-independent feature vectors, and an SVM or decision tree is trained to classify properties as fault-revealing or non-fault-revealing. The trained classifier is then applied to properties extracted from a previously unseen, potentially faulty program, to reveal properties that indicate latent errors.

Tractable Probabilistic Models

Sum-Product Networks

A sum-product network (SPN) is a rooted directed acyclic graph with univariate distributions at the leaves; the internal nodes are (weighted) sums and (unweighted) products.

Definition 1. (Gens and Domingos 2013)

1. A tractable univariate distribution is an SPN.

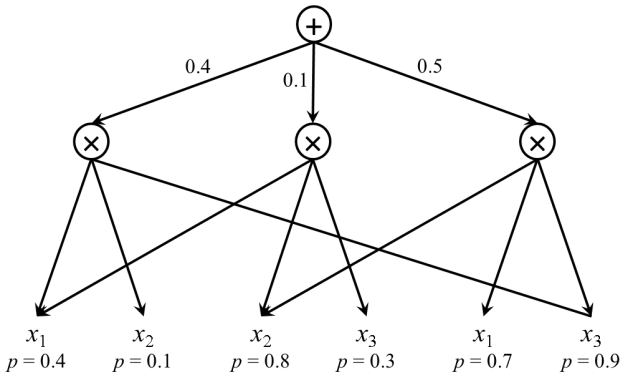

Figure 1: Example SPN over the variables x_1, x_2 and x_3. All leaves are Bernoulli distributions, with the given parameters. The weights of the sum node are indicated next to the corresponding edges.

2. A product of SPNs with disjoint scopes is an SPN. (The scope of an SPN is the set of variables that appear in it.)
3. A weighted sum of SPNs with the same scope is an SPN, provided all weights are positive.
4. Nothing else is an SPN.

Intuitively, an SPN (fig. 1) can be thought of as an alternating set of mixtures (sums) and decompositions (products) of the leaf variables. If the values at the leaf nodes are set to the partition functions of the corresponding univariate distributions, then the value at the root is the partition function (i.e. the sum of the unnormalized probabilities of all possible assignments to the leaf variables). This allows the partition function to be computed in time linear in the size of the SPN.

If the values of some variables are known, the leaves corresponding to those variables' distributions are set to those values' probabilities, and the remainder are replaced by their (univariate) partition functions. This yields the unnormalized probability of the evidence, which can be divided by the partition function to obtain the normalized probability. The most probable state of the SPN, viewing sums as marginalized-out hidden variables, can also be computed in linear time. The first learning algorithms for sum-product networks used a fixed network structure, and only optimized the weights (Poon and Domingos 2011; Amer and Todorovic 2012; Gens and Domingos 2012). More recently, several structure learning algorithms for SPNs have also been proposed (Dennis and Ventura 2012; Gens and Domingos 2013; Peharz, Geiger, and Pernkopf 2013).

Relational Sum-Product Networks

SPNs are a propositional representation, modeling instances as independent and identically distributed (i.i.d.). Although the i.i.d. assumption is widely used in statistical machine learning, it is often an unrealistic assumption. In practice, objects usually interact with each other; Statistical Relational Learning algorithms can capture dependencies between objects, and make predictions about relationships between them.

Relational Sum-Product Networks (RSPNs; Nath and Domingos 2015) generalize SPNs by modeling a set of instances jointly, allowing them to influence each other's probability distributions, as well as modeling probabilities of relations between objects. RSPNs can be seen as templates for constructing SPNs, much like Markov Logic Networks (Richardson and Domingos 2006) are templates for Markov networks. RSPNs also require as input a part decomposition, which describes the part-of relationships among the objects in the mega-example. Unlike previous high-treewidth tractable relational models such as TML (Domingos and Webb 2012), RSPNs can generalize across mega-examples of varying size and structure.

Tractable Fault Localization

Tractable Fault Localization Models

A *Tractable Fault Localization Model* (TFLM) defines a probability distribution over programs in some deterministic language L. The distribution may also model additional variables of interest that are not part of the program itself; we refer to such variables as *attributes*. In the fault localization setting, the important attribute is a *buggy* indicator variable on each line. Other informative features may also be included as attributes; for instance, one or more coverage-based metrics may be included for each line.

More formally, consider a language whose grammar $L = (V, \Sigma, R, S)$ consists of:

- V is a set of *non-terminal* symbols;
- Σ is a set of *terminal* symbols;
- R is a set of production rules of the form $\alpha \to \beta$, where $\alpha \in V$ and β is a string of symbols in $V \cup \Sigma$;
- $S \in V$ is the start symbol.

Definition 2. A *Tractable Fault Localization Model* for language L consists of:

- a map from non-terminals in V to sets of *attribute variables* (discrete or continuous);
- for each symbol $\alpha \in V$, a set of latent subclasses $\alpha_1, \ldots, \alpha_k$;
- π_S, a probability distribution over subclasses of the start symbol S;
- for each each subclass α_i of α,
  - a univariate distribution $\psi_{\alpha_i, x}$ over each attribute x associated with α;
  - for each rule $\alpha \to \beta$ in L, a probability distribution ρ_{α_i} over rules $\alpha_i \to \beta$;
  - for each rule $\alpha_i \to \beta$, for each non-terminal $\alpha' \in \beta$, a distribution $\pi_{(\alpha_i \to \beta), \alpha'}$ over subclasses of α'.

The univariate distribution over each attribute may be replaced with a joint distribution over all attributes, such as a logistic regression model within each subclass that predicts the value of the *buggy* attribute, using one or more other attributes as features. However, for simplicity, we present the remainder of this section with the attributes modeled as a product of univariate distributions, and assume that the attributes are discrete.

TFLMs are related to the Latent Variable PCFG (L-PCFG) models used in the natural language processing (NLP) community (Matsuzaki, Miyao, and Tsujii 2005; Prescher 2005; Petrov et al. 2006). As in L-PCFGs, each symbol α in the language is drawn probabilistically from a set of latent subclasses $\alpha_1, \ldots, \alpha_k$. For each latent class, the model can define a different distribution over the sub-trees rooted at that symbol.

Conceptually, TFLMs differ from the PCFG-based models as used in NLP in two ways:

1. In addition to modeling a distribution over strings in the given language, TFLMs can also jointly model other variables of interest ('attributes'). Different latent subclasses can have different distributions over attributes.
2. In NLP, PCFGs and their extensions are usually used for parsing, i.e. finding the most probably parse tree according to the given probabilistic grammar. In the debugging context, we assume the program can be parsed unambiguously. The purpose of a TFLM is to answer probabilistic queries about the attributes of the given program (e.g. infer marginal probabilities).

Being defined over the grammar of the programming language, TFLMs can capture information extremely useful for the fault localization task. For example, a TFLM can represent different fault probabilities for different symbols in the grammar. In addition, the latent subclasses give TFLMs a degree of context-sensitivity; the same symbol can be more or less likely to contain a fault depending on its latent subclass, which is probabilistically dependent on the subclasses of ancestor and descendent symbols in the parse tree. This makes TFLMs much richer than models like logistic regression, where the features are independent conditioned on the class variable. Despite this representational power, exact inference in TFLMs is still computationally efficient.

Example 1. The following rules are a fragment of the grammar of a Python-like language:

```
while_stmt → 'while' condition ':' suite
condition → expr operator expr
condition → 'not' condition
```

We refer to the above rules as r_1, r_2 and r_3 respectively.

The following is a partial specification of a TFLM over this grammar, with the `while_stmt` symbol as root.

- All non-terminal symbols have *buggy* and *suspiciousness* attributes. *buggy* is a fault indicator, and *suspiciousness* is a diagnostic attribute, such as a TARANTULA score.
- Each non-terminal has two latent subclass symbols. For example, `while_stmt` has subclasses $\texttt{while\_stmt}_1$ and $\texttt{while\_stmt}_2$.
- The distribution over start symbols is: $\pi(\texttt{while\_stmt}_1) = 0.4, \quad \pi(\texttt{while\_stmt}_2) = 0.6.$
- For subclass symbol $\texttt{while\_stmt}_1$ (subclass subscripts omitted):
  - $\psi_{buggy} \sim Bernoulli(0.01)$
  - $\psi_{suspiciousness} \sim \mathcal{N}(0.4, 0.05)$
  - $\rho(r_1) = 1.0$, since `while_stmt` has a single rule.
  - The distributions over child symbol subclasses for r_1 are: $\pi_{r_1,\texttt{condition}}(\texttt{condition}_1) = 0.7, \quad \pi_{r_1,\texttt{condition}}(\texttt{condition}_2) = 0.3,$ $\pi_{r_1,\texttt{suite}}(\texttt{suite}_1) = 0.2, \quad \pi_{r_1,\texttt{suite}}(\texttt{suite}_2) = 0.8.$

(The complete TFLM specification would have similar definitions for all the other subclass symbols in the model.)

Semantics Conceptually, a TFLM defines a probability distribution over all programs in L, and their attributes. More formally, the joint distribution $P(T, A, C)$ is defined over:

- a parse tree T;
- an attribute assignment A, specifying values of all attributes of all non-terminal symbols in T;
- latent subclass assignment C for each non-terminal in T.

For parse tree T containing rules $r_1 = \alpha_1 \rightarrow \beta_1, r_2 = \alpha_2 \rightarrow \beta_2, \ldots, r_n = \alpha_n \rightarrow \beta_n$, and root symbol α_R, $P(T, A, C) = \prod_{r_i} \left(\rho_{C(\alpha_i)}(\alpha_i \rightarrow \beta_i) \times \prod_{\alpha' \in \beta_i} \pi_{(C(\alpha_i)\rightarrow\beta_i),\alpha'}(C(\alpha')) \times \prod_{x \in attr(\alpha_i)} \psi_{C(\alpha_i),x}(A(x)) \right) \times \pi_S(C(\alpha_R))$

Inference Like RSPNs, inference in TFLMs is performed by grounding out the model into an SPN. The SPN is constructed in a recursive top-down manner, beginning with the start node:

- Emit a sum node over subclasses of the start node, weighted according to π_S. Let the current symbol α be the start symbol, and let α_i be its subclass. Let N^+_α refer to the sum node over subclasses of symbol α.
- Emit a product node $N^\times_{\alpha_i}$ with one child for each attribute of α, and a child for the subprogram rooted at α_i. ($N^\times_{\alpha_i}$ in turn is a child of N^+_α.)
- For each attribute x of α, emit univariate distribution $\psi_{\alpha_i,x}$ over the attribute values for the current symbol, as a child of $N^\times_{\alpha_i}$.
- Emit a sum node $N^+_{\rho_{\alpha_i}}$ over production rules $\alpha_i \rightarrow \beta$, weighted according to ρ_{α_i}. $N^+_{\rho_{\alpha_i}}$ is also a child of $N^\times_{\alpha_i}$. (Note that when grounding a TFLM over a known parse tree, all but one child of this sum node is zeroed out, and need not be grounded.)
- Recurse over each non-terminal α' in β, choosing its subclass via a sum node weighted by $\pi_{(\alpha_i\rightarrow\beta),\alpha'}$. The recursively constructed sub-SPNs are added as children to a product node $N^\times_{\alpha_i\rightarrow\beta}$, which in turn is a child of $N^+_{\rho_{\alpha_i}}$.

Following the above recursive grounding procedure naïvely results in a tree-structured SPN. However, note that we can cache the sub-SPN corresponding to each subclass of each symbol α in the parse tree, grounding it once and connecting it to multiple parents. The resulting SPN is a directed acyclic graph.

Table 1: Subject programs

Program	Versions	Executable LOC	Buggy vers.
grep	4	3368 ± 122	8 ± 5
gzip	5	1905 ± 124	7 ± 3
flex	5	3907 ± 254	10 ± 4
sed	7	2154 ± 389	3 ± 1

Learning The learning problem in TFLMs is to estimate the π and ψ distributions from a training corpus of programs, with known attribute values but unknown latent subclasses. (The ρ distributions have no effect on the distribution of interest, since we assume that every program can be unambiguously mapped to a parse tree.)

As is commonly done with SPNs, we train the model via hard EM. In the E-step, given the current parameters of π and ψ, we compute the MAP state of the training programs (i.e. the latent subclass assignment that maximizes the log-probability). In the M-step, we re-estimate the parameters of π and ψ, choosing the values that maximize the log-probability. These two steps are repeated until convergence, or for a fixed number of iterations.

If the attributes are modeled jointly rather than as a product of univariate distributions, retraining the joint model in each iteration of EM is computationally expensive. A more efficient alternative is to use a product of univariates during EM, in order to learn a good subclass assignment. The joint model is then only trained once, at the conclusion of EM.

Experiments

We performed an experiment to determine whether TFLM's ability to combine a coverage-based fault localization system with learned bug patterns improves fault localization performance, relative to using the coverage-based system directly. As a representative coverage-based method, our study used TARANTULA, one of the most widely-used approaches in this class, and a common comparison system for fault localization algorithms. We also compared to the statement-based version of Liblit et al.'s Statistical Bug Isolation (SBI) system (Liblit et al. 2005), as adapted by Yu et al. (2008). SBI serves as a representative example of a lightweight statistical method for fault localization.

Subjects

We evaluated TFLMs on four mid-sized C programs (table 1) from the Software-artifact Infrastructure Repository (Khurshid et al. 2004). All four test subjects are real-world programs, commonly used to evaluate fault localization approaches. The repository contained several sequential versions of each program, each with several buggy versions. The repository also contained a suite of between 124 and 525 TSL tests for each version, which we used to compute the TARANTULA scores.

The number of executable lines was measured by the gcov tool. We excluded buggy versions where the bug occurred in a non-executable line (e.g. lines excluded by preprocessor directives), or consisted of line insertions or deletions. Unlike most previous fault localization studies that use these subjects, we do not exclude versions for which the test results were uniform (i.e. consisting entirely of passing or failing tests). Although coverage-only methods such as TARANTULA can provide no useful information in the case of uniform test suites, TFLMs can still make use of learned contextual information to determine that some lines are more likely than others to contain a fault.

Table 2: Localization accuracy (fraction of lines skipped)

Program	TFLM	Tarantula	SBI
grep	**0.645**	0.640	0.564
gzip	0.516	**0.682**	0.540
flex	**0.770**	0.704	0.618
sed	**0.927**	0.851	0.603

Table 3: TFLM (with Tarantula feature) vs Tarantula alone

Program	TFLM wins	Ties	Tarantula wins
grep	**18**	0	14
gzip	11	0	**23**
flex	**31**	0	18
sed	**17**	0	7

Methodology

We implemented TFLMs for a simplified version of the C grammar with 23 non-terminal symbols, ranging from compound statements like `if` and `while` to atomic single-line statements such as assignments and `break` and `continue` statements. Each symbol has a *buggy* attribute, and a *suspiciousness* attribute, which is the TARANTULA score of the corresponding line. (For AST nodes that correspond to multiple lines in the original code, we use the highest TARANTULA score among all lines). As described in the previous section, the attributes are modeled as independent univariates during EM (*buggy* as a Bernoulli distribution, and *suspiciousness* as a Gaussian), and then via a logistic regression model within each subclass. The model predicts the *buggy* attribute, using the TARANTULA score and a bias term as features. We use the SCIKIT-LEARN (Pedregosa et al. 2011) implementation of logistic regression, with the `class_weight='auto'` parameter, to compensate for the sparsity of the buggy lines relative to bug-free lines. For TFLMs, we ran hard EM for 100 iterations. For each subject program, TFLMs were learned via cross-validation, training on all versions of the program except the one being evaluated. The number of latent subclasses was also chosen via cross-validation, from the range $[1, 4]$.

The output of a fault localization system is a ranking of

Table 4: TFLM learning and inference times

Program	Avg. learn time (s)	Avg. infer time (s)
grep	1135.00	20.91
gzip	433.33	5.25
flex	978.63	13.15
sed	326.37	5.18

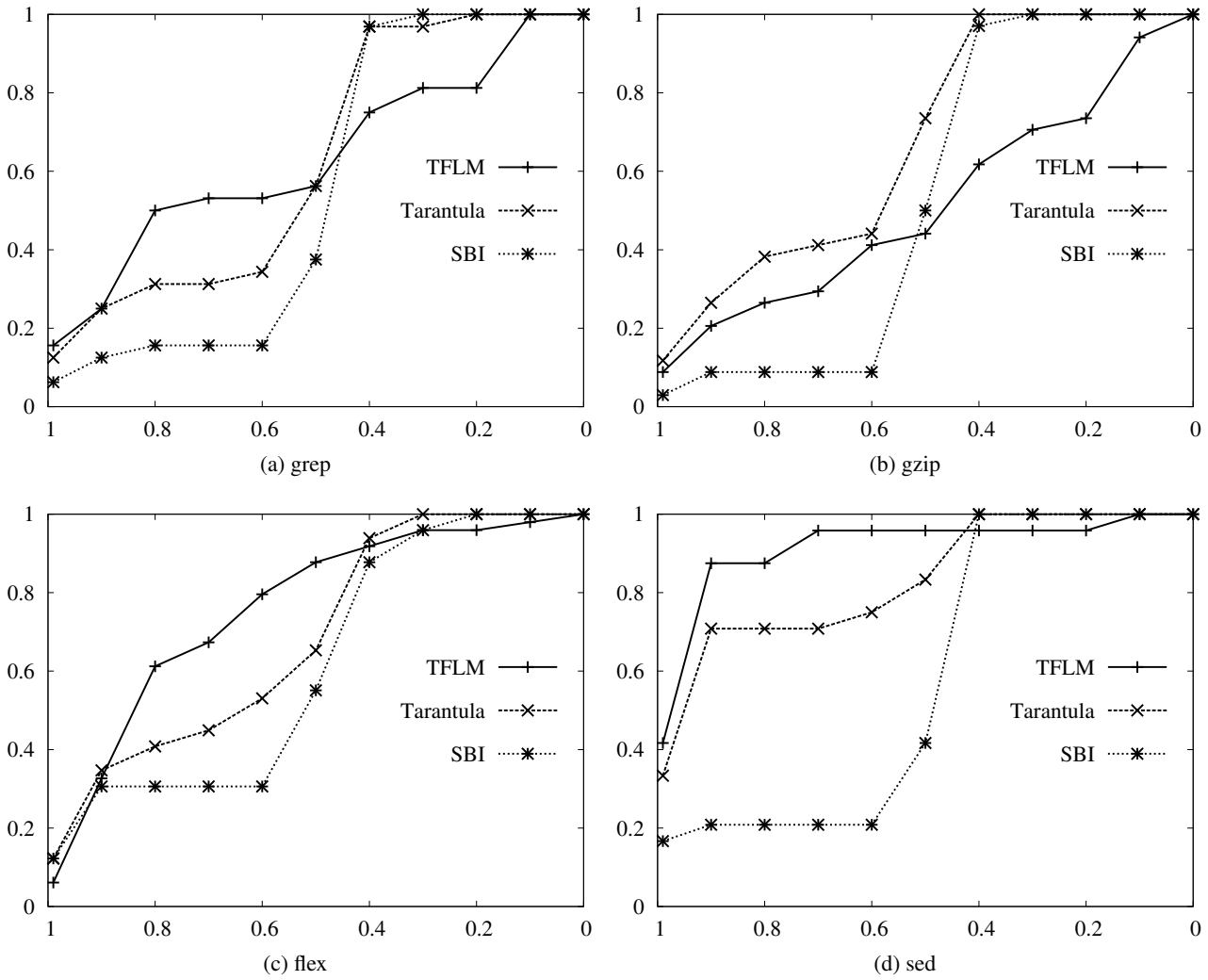

Figure 2: The horizontal axis is the fraction of lines skipped (FS), and the vertical axis is the fraction of runs for which the FS score equalled or exceeded the x-axis value.

the lines of code from most to least suspicious. For TFLMs, we ranked the lines by predicted probability that *buggy* $= 1$. (Each line in the original program is modeled by the finest-grained AST node that encloses it.) The evaluation metric was the 'fraction skipped' (FS) score, i.e. the fraction of executable lines ranked below the highest-ranked buggy line. Despite its limitations (Parnin and Orso 2011), this is a widely-used metric for fault localization (Jones and Harrold 2005; Abreu et al. 2009).

Results

Results of our experiments are displayed in tables 2 and 3, and figure 2. TFLMs outperform TARANTULA and SBI on three of the four subjects, isolating the majority of bugs more effectively, and earning a higher average FS score. However, TFLMs perform poorly on the `gzip` domain. This demonstrates the main threat to the validity of our method: machine learning algorithms operate under the assumption that the test data is drawn from a similar distribution to the training data. If the bugs occur in different contexts in the training and test datasets (as in `gzip`), learning-based methods may perform worse than methods that try to localize each program independently. This risk is particularly great when the learning from a small corpus of buggy programs.

However, in three of the four subjects in our experiment, the training and test distributions are sufficiently similar to allow useful generalization, resulting in improved fault localization performance. TFLMs' advantage arises from its ability to localize faults even when the coverage matrix used by TARANTULA does not provide useful information (e.g. when the tests are not sufficiently discriminative). TFLMs combine the coverage-based information used by TARANTULA with learned bug probabilities for different symbols, in different contexts. Context sensitivity is captured via latent subclass assignments for each symbol.

As seen in table 4, our unoptimized Python implementa-

tion predicts bug probabilities in a few seconds for programs a few thousand lines in length. An optimized implementation may be able to make predictions at interactive speeds; this makes TFLMs a practical choice of inference engine for a debugging tool in a software development environment. Learning TFLMs can take several minutes, but note that the model can be trained offline, either from previous versions of the software being developed (as in our experiments), or from other related software projects expected to have a similar bug distribution (e.g. projects of a similar scale, written in the same language).

Conclusions

This paper presented TFLMs, a probabilistic model for fault localization that can be learned from a corpus of buggy programs. This allows the model to generalize from previously seen bugs to more accurately localize faults in a new context. TFLMs can also incorporate the output of other fault-localization systems as features in the probabilistic model, with a learned weight that depends on the context. TFLMs take advantage of recent advances in tractable probabilistic models to ensure that the fault location probabilities can be inferred efficiently even as the size of the program grows. In our experiments, a TFLM trained with TARANTULA as a feature localized bugs more effectively than TARANTULA or SBI alone, on three of the four subject programs.

In this work, we used TFLMs to generalize across sequential versions of a single program. Given adequate training data, TFLMs could also be used to generalize across more distantly-related programs. The success of this approach relies on the assumption that there is some regularity in software faults, i.e. the same kinds of errors occur repeatedly in unrelated software projects, with sufficient regularity that a machine learning algorithm can generalize over these programs. Testing this assumption is a direction for future work.

Another direction for future work is extending TFLMs with additional sources of information, such as including multiple fault localization systems, and richer program features derived from static or dynamic analysis (e.g. invariants (Hangal and Lam 2002; Brun and Ernst 2004)). TFLM-like models may also be applicable to debugging methods that use path profiling (Chilimbi et al. 2009), giving the user more contextual information about the bug, rather than just a ranked list of statements. The recent developments in tractable probabilistic models may also enable advances in other software engineering problems, such as fault correction, code completion, and program synthesis.

Acknowledgments

This research was partly funded by ONR grants N00014-13-1-0720 and N00014-12-1-0312, and AFRL contract FA8750-13-2-0019. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ONR, AFRL, or the United States Government.

References

Abreu, R.; Zoeteweij, P.; Golsteijn, R.; and van Gemund, A. J. 2009. A practical evaluation of spectrum-based fault localization. *The Journal of Systems and Software* 82(11):1780–1792.

Amer, M. R., and Todorovic, S. 2012. Sum-product networks for modeling activities with stochastic structure. In *CVPR*.

Andrzejewski, D.; Mulhern, A.; Liblit, B.; and Zhu, X. 2007. Statistical debugging using latent topic models. In *ECML/PKDD*.

Ascari, L. C.; Araki, L. Y.; Pozo, A. R. T.; ; and Vergilio, S. R. 2009. Exploring machine learning techniques for fault localization. In *Latin American Test Workshop*.

Briand, L. C.; Labiche, Y.; and Liu, X. 2007. Using machine learning to support debugging with TARANTULA. In *ISSRE*.

Brun, Y., and Ernst, M. 2004. Finding latent code errors via machine learning over program executions. In *ICSE*.

Chilimbi, T. M.; Liblit, B.; Mehra, K.; Nori, A. V.; and Vaswani, K. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *ICSE*.

Cleve, H., and Zeller, A. 2005. Locating causes of program failures. In *ICSE*.

Dennis, A., and Ventura, D. 2012. Learning the architecture of sum-product networks using clustering on variables. In *NIPS*.

Domingos, P., and Webb, A. 2012. A tractable first-order probabilistic logic. In *AAAI*.

Ernst, M. D.; Cockrell, J.; Griswold, W. G.; and Notkin, D. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27(2):99–123.

Gens, R., and Domingos, P. 2012. Discriminative learning of sum-product networks. In *NIPS*.

Gens, R., and Domingos, P. 2013. Learning the structure of sum-product networks. In *ICML*.

Hailpern, B., and Santhanam, P. 2002. Software debugging, testing, and verification. *IBM Systems Journal* 41(1):4–12.

Hangal, S., and Lam, M. S. 2002. Tracking down software bugs using automatic anomaly detection. In *ICSE*.

Jones, J. A., and Harrold, M. J. 2005. Empirical evaluation of the TARANTULA automatic fault-localization technique. In *ASE*.

Jones, J. A.; Harrold, M. J.; and Stasko, J. 2002. Visualization of test information to assist fault localization. In *ICSE*.

Khurshid, S.; Marinov, D.; Rothermel, G.; Xie, T.; and Motycka, W. 2004. Software-artifact infrastructure repository (SIR). Retrieved 15 August, 2014 from http://sir.unl.edu/.

Liblit, B.; Aiken, A.; Zheng, A. X.; and Jordan, M. I. 2003. Bug isolation via remote program sampling. In *PLDI*.

Liblit, B.; Naik, M.; Zheng, A. X.; Aiken, A.; and Jordan, M. I. 2005. Scalable statistical bug isolation. In *PLDI*.

Liu, C.; Yan, X.; Fei, L.; Han, J.; and Midkiff, S. P. 2005. SOBER: Statistical model-based bug localization. In *ESEC/FSE*.

Liu, C.; Fei, L.; Yan, X.; Han, J.; and Midkiff, S. P. 2006. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering* 32(10):831–848.

Matsuzaki, T.; Miyao, Y.; and Tsujii, J. 2005. Probabilistic CFG with latent annotations. In *ACL*.

Nath, A., and Domingos, P. 2015. Learning relational sum-product networks. In *AAAI*.

Parnin, C., and Orso, A. 2011. Are automated debugging techniques actually helping programmers? In *ISSTA*.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; and Duchesnay, E. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12:2825–2830.

Peharz, R.; Geiger, B. C.; and Pernkopf, F. 2013. Greedy part-wise learning of sum-product networks. In *ECML/PKDD*.

Petrov, S.; Barrett, L.; Thibaux, R.; and Klein, D. 2006. Learning accurate, compact, and interpretable tree annotation. In *ACL/COLING*.

Poon, H., and Domingos, P. 2011. Sum-product networks: A new deep architecture. In *UAI*.

Prescher, D. 2005. Inducing head-driven PCFGs with latent heads: Refining a tree-bank grammar for parsing. In *ECML/PKDD*.

Renieris, M., and Reiss, S. P. 2003. Fault localization with nearest neighbor queries. In *ASE*.

Richardson, M., and Domingos, P. 2006. Markov logic networks. *Machine Learning* 62:107–136.

RTI International. 2002. The economic impacts of inadequate infrastructure for software testing. Retrieved 4 August, 2014 from http://www.nist.gov/director/planning/upload/report02-3.pdf.

Wong, W. E., and Qi, Y. 2009. BP neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering* 19(4):573–597.

Wong, W. E.; Wei, T.; Qi, Y.; and Zhao, L. 2008. A crosstab-based statistical method for effective fault localization. In *ICST*.

Wong, W. E.; Debroy, V.; Golden, R.; Xu, X.; and Thuraisingham, B. 2012. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability* 61(1):149–169.

Wong, W. E.; Debroy, V.; and Xu, D. 2012. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews* 42(3):378–396.

Yu, Y.; Jones, J. A.; and Harrold, M. J. 2008. An empirical study of the effects of test-suite reduction on fault localization. In *ICSE*.

Zhang, Z.; Chanb, W.; Tsec, T.; Yub, Y.; and Hu, P. 2011. Non-parametric statistical fault localization. *Journal of Systems and Software* 84(6):885–905.