Network-Clustered Multi-Modal Bug Localization

Van Duc Thong Hoang, Richard J. Oentaryo, Tien-Duy B. Le, and David Lo
School of Information Systems, Singapore Management University
{vdthoang.2016, roentaryo, btdle.2012, davidlo}@smu.edu.sg

Abstract—Developers often spend much effort and resources to debug a program. To help the developers, numerous information retrieval (IR)-based and spectrum-based bug localization techniques have thus been devised. IR-based techniques process textual information in bug reports, while spectrum-based techniques process program spectra (i.e., a record of which program elements are executed for each test case). While both techniques ultimately generate a ranked list of program elements that likely contain a bug, they only consider one source of information—either bug reports or program spectra—which is not optimal. In light of this deficiency, this paper presents a new approach dubbed Network-clustered Multi-Modal Bug Localization (NetML), which utilizes multi-modal information from both bug reports and program spectra to localize bugs. NetML facilitates an effective bug localization by performing joint optimization of localization quality and clustering of both bug reports and program elements. The clustering is achieved through the incorporation of *network Lasso* regularization, which incentivizes the latent parameters of similar bug reports and similar program elements to be close together. In addition to using bug report text and program spectra features (which are akin to the IR-based and spectra-based techniques respectively), NetML also incorporates a hybrid *method suspiciousness* feature that stems from processing both bug report and program spectra information. Extensive experiments on 157 real bugs from four software systems have been conducted to evaluate NetML against various state-of-the-art localization methods. The results show that NetML surpasses the best-performing baseline by at least 48.39%, 15.49%, 8.7%, and 13.92%. in terms of the number of bugs successfully localized when a developer inspects the top 1, 5, and 10 program elements and Mean Average Precision (MAP), respectively.

1 Introduction

Debugging bug reports, which often come in a high volume [9], has proved to be a difficult task that consumes much resources and time [53]. Various techniques have thus been devised to help developers locate buggy program elements from their symptoms. These symptoms could be in the form of a descriptions of a bug experienced by a user, or a failing test case. These techniques, often collectively referred to as bug (or fault) localization, would analyze the symptoms of a bug and produce a list of program elements ranked based on their likelihood to contain the bug.

Existing bug localization techniques broadly fall into two major categories: *information retrieval* (IR)-based techniques [44], [50], [65], [47], and *spectrum*-based bug localization techniques [22], [8], [46], [63], [64], [14], [27], [28], [29]. The IR-based bug localization techniques typically analyze textual descriptions contained in bug reports and identifier names and comments in source code files. They then return a ranked list of program elements (typically program files) that are the most similar to the bug textual description. The spectrum-based bug localization techniques typically analyze program spectra that corresponds to program elements that are executed by failing and successful execution traces. Likewise, they return a ranked list of program elements (typically program blocks or statements) that are executed more often in the failing rather than correct traces.

The above-mentioned approaches, however, only consider one kind of symptom or one source of information, i.e., only bug reports or only execution traces. This is a limiting factor since hints of the location of a bug may be spread in both bug report and execution traces; and some hints may only appear in one but not the other. In this work, we put forward a bug localization approach that addresses the deficiency of existing methods by jointly utilizing both bug reports and execution traces. We refer to this approach as *multi-modal bug localization*, as we need to consider multiple modes of inputs (i.e., bug reports and program spectra). Such an approach fits well to developers' debugging activities as illustrated by the following scenarios:

1) Developer D is working on a bug report that is submitted to Bugzilla. One of the first tasks that he needs to do is to replicate the bug based on the description in the report. If the bug can be successfully replicated, he will proceed to the debugging step; otherwise, he will mark the bug report as "WORKSFORME" and will not continue further [37]. After D replicates the bug, he has one or a few failing execution traces. He also has a set of regression tests that he can run to get successful execution traces. Thus, after the replication process, D has *both* the textual description of the bug and a program spectra that characterizes the bug. With this, D can proceed to use multi-modal bug localization.
2) Developer D runs a regression test suite and some test cases fail. Based on his experience, D has some idea why the test cases fail. D can create a textual document describing the bug. At the end of this step, D has *both* program spectra and textual bug description, and can proceed to use multi-modal bug localization which will leverage not only the program spectra but also D's domain knowledge to locate the bug.

It is worth noting that our work focuses on localizing a bug to the *method* that contains it. Historically, most IR-based bug localization techniques aim at finding buggy files [44],

[50], [65], [47], while most spectrum-based techniques find buggy lines [22], [8], [46]. Although it is useful to localize a bug to the file that contains it, the file size can be big and developers still need to go through a lot of code to find the few lines that contain the bug. On the other hand, while localizing a bug to the line that contains it is also useful, a bug often spans across multiple lines. Furthermore, developers often do not have "perfect bug understanding" [40] and thus by just looking at a line of code, developers often cannot determine whether it is the location of the bug and/or understand the bug well enough to fix it. Localization at the method level presents a good tradeoff; a method is not as big as a file, but it often contains sufficient context needed to help developers understand a bug.

In this paper, we present a new approach called the Network-clustered Multi-modal Bug Localization (NetML), which works based on three main intuitions. Firstly, it is established that a wide variety of bugs exist [54], [57], and different bugs often require different treatments. Hence, there is a need to have separate latent parameters for different bugs, which are tailored to the characteristics of the individual bugs. Similarly, different program elements (or methods in this work) are of different nature, and should be characterized using separate latent parameters. Secondly, Parnin and Orso [40] found in their studies that some words are more useful in localizing bugs, and suggested that "future research could also investigate ways to automatically suggest or highlight terms that might be related to a failure". NetML provides such capability by incorporating *method suspiciousness* feature, which allows us to automatically highlight suspicious terms and use them to localize bugs. Lastly, we recognize that bugs and program elements are *not* completely independent, and some bugs (or methods) may be more similar to certain bugs (or methods) than to others. As such, similar bugs (or methods) should have latent parameters that are close together, which can then be exploited to localize bugs more effectively.

Some of these intuitions have already been captured in our recent work—dubbed AML [25]. In particular, AML already incorporates the ideas of adaptively computing separate latent parameters for each bug report, and of computing the method suspiciousness score. However, the current AML approach exhibits two main shortcomings. Firstly, AML only has the concept of latent parameters for bug reports, but not for program elements (or methods). As such, it is not able to capture the variation in the (latent) characteristics of different program elements (methods), which may limit its effectiveness in localizing a bug. Secondly, the latent parameters of each bug report are learned independently of those of other bug reports. As a result, AML is unable to take advantage of the clustering/similarity traits of different bug reports in the localization process.

The proposed NetML method addresses these shortcomings by performing joint optimization of localization loss function and clustering of both bug reports and methods. Specifically, it generalizes AML in two ways. Firstly, NetML features two sets of latent parameters—one for bug reports and the other for methods. The incorporation of the additional latent method parameters provides NetML with a higher degree of freedom to model the rich variations of different bug reports and methods, which would enable it to achieve more accurate bug localization. Secondly,

We evaluate our proposed approach (i.e., N-AML) using a dataset of 157 real bugs from four medium to large software systems: AspectJ, Ant, Lucene, and Rhino. All real bug reports and real test cases are collected from these systems. The test cases are run to generate program spectra. We compare N-AML with original AML. Moreover, we also compare our approach against 3 state-of-the-art multi-modal feature localization techniques (i.e., PROMESIR [42], DIT^A and DIT^B [18]), a state-of-the-art IR-based bug localization technique [61], and a state-of-the-art spectrum-based bug localization technique [60]. We use two well-known evaluation metrics to estimate the performance of our approach: number of bugs localized by inspecting the top N program elements (Top N) and mean average precision (MAP). Top N and MAP are widely used in past bug localization studies, e.g., [45], [51], [65], [47]. Top N is in line with the observation of Parnin and Orso, who highlight that developers care about absolute rank and they often will stop inspecting program elements, if they do not get promising results when inspecting the top ranked program elements [40]. MAP is a standard information retrieval metric to evaluate the effectiveness of a ranking technique [35].

Our experiment results highlight that, among the 157 bugs, -N-AML can successfully localize 46, 82, and 100 bugs when developers only inspect the top 1, top 5, and top 10 methods in the lists that N-AML produces respectively. AML can successfully localize 47.62%, 31.48%, and 27.78% more bugs than the best baseline when developers only inspect the top 1, top 5, and top 10 methods, respectively. In terms of MAP, AML outperforms the best performing baseline by 28.80%. AML* also outperforms the best baselines by at least 119.05%, 51.85%, 38.89%, and 46.74% in terms of top 1, top 5 , top 10, and MAP respectively. Moreover, AML* also outperforms AML by 48.39%, 15.49%, 8.7%, and 13.92% when the developer investigates top 1, top 5, top 10 methods and MAP respectively.

We summarize our key contributions (inclusive of our conference paper [25]) below:

1) We are the first to build an adaptive algorithm for multi-modal bug localization. Different from past approaches which are one-size-fits-all, our approach is instance-specific and considers each individual bug to tune various parameters or weights of components (i.e., AML$^{\text{Text}}$, AML$^{\text{SuspWord}}$, and AML$^{\text{Spectra}}$).
2) We are the first to compute suspicious words and use these words to help bug localization. Past studies only compute suspiciousness scores of program elements.
3) We develop a probabilistic-based iterative optimization procedure to find the best linear combination of AML components (i.e., AML$^{\text{Text}}$, AML$^{\text{SuspWord}}$, and AML$^{\text{Spectra}}$) that maximizes the posterior probability of bug localization. The procedure features an efficient and balanced sampling strategy to gracefully handle the skewed distribution of the faulty vs. non-faulty methods (i.e., given a bug, there are more non-faulty methods than faulty ones in a code base).
4) We propose a generalized procedure that further optimizes the combinations of AML components (i.e., AML$^{\text{Text}}$, AML$^{\text{SuspWord}}$, and AML$^{\text{Spectra}}$), and call it as AML*. The procedure assigns a distinctive tuple of

weights for every method and takes into account the relationship of bug reports and program methods. Duy: please modify if needed.

5) We have evaluated AML* on 157 real bugs from 4 software systems using real bug reports and test cases. Our experiments highlight that our proposed approach improves upon state-of-the-art multi-modal bug localization solutions by a substantial margin.

The remainder of this paper is organizes as follows. In Section 2, we present background information on IR-based and spectrum-based bug localization approaches. Section 3 subsequently elaborates the proposed NetML in greater details. In Section 4, we present our dataset, evaluation metrics, and experiment results. Section 5 then provides an overview of key related works. We finally conclude and discuss future works in Section 6.

2 Background

In this section, we present some background material on IR-based and spectrum-based bug localization.

IR-Based Bug Localization: IR-based bug localization techniques consider an input bug report (i.e., the text in the summary and description of the bug report – see Figure 1) as a query, and program elements in a code base as documents, and employ information retrieval techniques to sort the program elements based on their relevance with the query. The intuition behind these techniques is that program elements sharing many common words with the input bug report are likely to be relevant to the bug. By using text retrieval models, IR-based bug localization computes the similarities between various program elements and the input bug report. Then, program elements are sorted in descending order of their textual similarities to the bug report, and sent to developers for manual inspection.

All IR-based bug localization techniques need to extract textual contents from source code files and preprocess textual contents (either from bug reports or source code files). First, comments and identifier names are extracted from source code files. These can be extracted by employing a simple parser. In this work, we use JDT [5] to recover the comments and identifier names from source code. Next, after the textual contents from source code and bug reports are obtained, we need to preprocess them. The purpose of text preprocessing is to standardize words in source code and bug reports. There are three main steps: text normalization, stopword removal, and stemming:

1) Text normalization breaks an identifier into its constituent words (tokens), following camel casing convention. Following the work by Saha et al. [47], we also keep the original identifier names.
2) Stopword removal removes punctuation marks, special symbols, number literals, and common English stopwords [6]. It also removes programming keywords such as *if*, *for*, *while*, etc., as these words appear too frequently to be useful enough to differentiate between documents.
3) Stemming simplifies English words into their root forms. For example, "processed", "processing", and "processes" are all simplified to "process". This increases the chance of a query and a document to share some common words. We use the popular Porter Stemming algorithm [41].

Bug 54460

`Summary`: Base64Converter not properly handling bytes with MSB set (not masking byte to int conversion)

`Description`: Every 3rd byte taken for conversion (least significant in triplet is not being masked with added to integer, if the msb is set this leads to a signed extension which overwrites the previous two bytes with all ones . . .

Fig. 1. Bug Report 54460 of Apache Ant

There are many IR techniques that have been employed for bug localization. We highlight a popular IR technique namely *Vector Space Model* (VSM). In VSM, queries and documents are represented as vectors of weights, where each weight corresponds to a term. The value of each weight is usually the *term frequency—inverse document frequency* (TF-IDF) of the corresponding word. Term frequency refers to the number of times a word appears in a document. Inverse document frequency refers to the number of documents in a corpus (i.e., a collection of documents) that contain the word. The higher the term frequency and inverse document frequency of a word, the more important the word would be. In this work, given a document d and a corpus C, we compute the TF-IDF weight of a word w as follows:

$$\text{TF-IDF}(w,d,C) = \log(f(w,d)+1) \times \log \frac{|C|}{|d_i \in C : w \in d_i|}$$

where $f(w,d)$ is the number of times word w appears in document d.

After computing a vector of weights for the query and each document in the corpus, we calculate the cosine similarity of the query's vector and the document's vector. The cosine similarity between query q and document d is given by:

$$sim(q,d) = \frac{\sum\limits_{w \in (q \bigcap d)} weight(w,q) \times weight(w,d)}{\sqrt{\sum\limits_{w \in q} weight(w,q)^2} \times \sqrt{\sum\limits_{w \in d} weight(w,d)^2}}$$

where $w \in (q \bigcap d)$ means word w appears both in the query q and document d. Also, $weight(w,q)$ refers to the weight of word w in the query q's vector. Similarly, $weight(w,d)$ refers to the weight of word w in the document d's vector.

Spectrum-Based Bug Localization: Spectrum-based bug localization (SBBL), also known as spectrum-based fault localization (SBFL), takes as input a faulty program and two sets of test cases. One is a set of failed test cases, and the other one is a set of passed test cases. SBBL then instruments the target program, and records program spectra that are collected when the set of failed and passed test cases are run on the instrumented program. Each of the collected program spectrum contains information of program elements that are executed by a test case. Various tools can be used to collect program spectra as a set of test cases are run. In this work, we use Cobertura [4].

TABLE 1
Raw Statistics for Program Element e

	e is *executed*	e is *not executed*
unsuccessful test	$n_f(e)$	$n_f(\bar{e})$
successful test	$n_s(e)$	$n_s(\bar{e})$

TABLE 2
Raw Statistic Description

Notation	**Description**
$n_f(e)$	Number of unsuccessful test cases that execute program element e
$n_f(\bar{e})$	Number of unsuccessful test cases that do not execute program element e
$n_s(e)$	Number of successful test cases that execute program element e
$n_s(\bar{e})$	Number of successful test cases that do not execute program element e
n_f	Total number of unsuccessful test cases
n_s	Total number of successful test cases

Based on these spectra, SBBL typically computes some raw statistics for every program elements. Tables 1 and 2 summarize some raw statistics that can be computed for a program element e. These statistics are the counts of unsuccessful (i.e., failed), and successful (i.e., passed) test cases that execute or do not execute e. If a successful test case executes program element e, then we increase $n_s(e)$ by one unit. Similarly, if an unsuccessful test case executes program element e, then we increase $n_f(e)$ by one unit. SBBL uses these statistics to calculate the suspiciousness scores of each program element. The higher the suspiciousness score, the more likely the corresponding program element is the faulty element. After the suspiciousness scores of all program elements are computed, program elements are then sorted in descending order of their suspiciousness scores, and sent to developers for manual inspection.

There are a number of SBBL techniques which propose various formulas to calculate suspiciousness scores. Among these techniques, Tarantula is a popular one [22]. Using the notation in Table 2, the following is the formula that Tarantula uses to compute the suspiciousness score of program element e:

$$Tarantula(e) = \frac{\frac{n_f(e)}{n_f}}{\frac{n_f(e)}{n_f} + \frac{n_s(e)}{n_s}}$$

The main idea of Tarantula is that program elements that are executed by failed test cases are more likely to be faulty than the ones that are not executed by failed test cases. Thus, Tarantula assigns a non-zero score to program element e that has $n_f(e) > 0$.

3 Proposed Approach

The overall framework of our Adaptive Multi-modal bug Localization (AML) is shown in Figure 2. AML (enclosed in dashed box) takes as input a new bug report and the program spectra corresponding to it. AML also takes as input a training set of (historical) bugs that have been localized before. For each bug in the training set, we have its bug report, program spectra, and set of faulty methods.

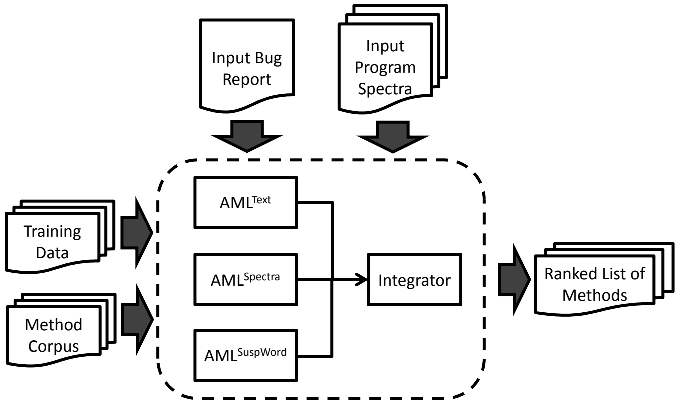

Fig. 2. Framework of our proposed bug localization approach

If a method contains a root cause of the bug, it is labeled as *faulty*, otherwise it is labeled as *non-faulty*. Based on the training set of previously localized bugs and a method corpus, AML produces a list of methods ranked based on their likelihood to be the faulty ones given the new bug report.

AML has four components: AML$^{\text{Text}}$, AML$^{\text{Spectra}}$, AML$^{\text{SuspWord}}$, and Integrator. AML$^{\text{Text}}$ processes only the textual information in the input bug reports using an IR-based bug localization technique described in Section 2. AML$^{\text{Text}}$ in the end outputs a score for each method in the corpus. Given a bug report b and a method m in a corpus C, AML$^{\text{Text}}$ outputs a score that indicates how close is m to b which is denoted as AML$^{\text{Text}}(b, m, C)$. By default, AML$^{\text{Text}}$ uses VSM as the IR-based bug localization technique.

AML$^{\text{Spectra}}$ processes only the program spectra information using a spectrum-based bug localization technique described in Section 2. AML$^{\text{Spectra}}$ in the end outputs a score for each method in the corpus. Given a program spectra p and a method m in a corpus C, AML$^{\text{Spectra}}$ outputs a score that indicates how suspicious is m considering p which is denoted as AML$^{\text{Spectra}}(p, m, C)$. By default, AML$^{\text{Spectra}}$ uses Tarantula as the spectrum-based bug localization technique.

AML$^{\text{SuspWord}}$ processes both bug reports and program spectra, and computes the suspiciousness scores of words to rank methods. Given a bug report b, a program spectra p, and a method m in a corpus C, AML$^{\text{SuspWord}}$ outputs a score that indicates how suspicious is m considering b and p; this is denoted as AML$^{\text{SuspWord}}(b, p, m, C)$.

The integrator component combines the AML$^{\text{Text}}$, AML$^{\text{Spectra}}$, AML$^{\text{SuspWord}}$ components to produce the final ranked list of methods. Given a bug report b, a program spectra p, and a method m in a corpus C, the adaptive integrator component outputs a suspiciousness score for method m which is denoted as AML(b, p, m, C).

The AML$^{\text{Text}}$ and AML$^{\text{Spectra}}$ components reuse techniques proposed in prior works which are described in Section 2. In the next subsections, we just describe the new components namely AML$^{\text{SuspWord}}$ and the adaptive integrator component.

3.1 Suspicious Word Component

Parnin and Orso highlighted that "future research could also investigate ways to automatically suggest or highlight terms that might be related to a failure" [40], however they did not propose a concrete solution. We use Parnin and Orso's observation, which highlights that some words are indicative to the location of a bug, as a starting point to design our AML$^{\text{SuspWord}}$ component. This component breaks down a method into its constituent words, computes the suspiciousness scores of these words, and composes these scores back to result in the suspiciousness score of the method. The process is analogous to a machine learning or classification algorithm that breaks a data point into its constituent features, assign weights or importance to these features, and use these features, especially important ones, to assign likelihood scores to the data point. The component works in three steps: mapping of methods to words, computing word suspiciousness, and composing word suspiciousness into method suspiciousness. We describe each of these steps in the following paragraphs.

3.1.0.1 Step 1: Mapping of Methods to Words: In this step, we map a method to its constituent words. For every method, we extract the following textual contents including: (1) The name of the method, along with the names of its parameters, and identifiers contained in the method body; (2) The name of the class containing the method, and the package containing the class; (3) The comments that are associated to the method (e.g., the Javadoc comment of that method, and the comments that appear inside the method), and comments that appear in the class (containing the method) that are not associated to any particular method.

After we have extracted the above textual contents, we apply the text pre-processing step described in Section 2. At the end of this step, for every method we map it to a set of pre-processed words. Given a method m, we denote the set of words it contains as $words(m)$.

3.1.0.2 Step 2: Computing Word Suspiciousness: We compute the suspiciousness score of a word by considering the program elements that contain the word. Let us denote the set of all failing execution traces in spectra p as $p.F$ and the set of all successful execution traces as $p.S$. To compute the suspiciousness scores of a word w given spectra p, we define several sets:

$$EF(w,p) = \{t \in p.F | \exists m \in t \; s.t. \; w \in words(m)\}$$
$$ES(w,p) = \{t \in p.S | \exists m \in t \; s.t. \; w \in words(m)\}$$

The set $EF(w,p)$ is the set of execution traces in $p.F$ that contain a method in which the word w appears. The set $ES(w,p)$ is the set of execution traces in $p.S$ that contain a method in which the word w appears. Based on these sets, we can compute the suspiciousness score of a word w using a formula similar to Tarantula as follows:

$$\text{SS}_{\text{word}}(w,p) = \frac{\frac{|EF(w,p)|}{|p.FAIL|}}{\frac{|EF(w,p)|}{|p.FAIL|} + \frac{|ES(w,p)|}{|p.SUCCESS|}} \quad (1)$$

Using the above formula, words that appear more often in methods that are executed in failing execution traces are deemed to be more suspicious than those that appear less often in such methods.

3.1.0.3 Step 3: Computing Method Suspiciousness: To compute a method m's suspiciousness score, we compute the textual similarity between m and the input bug report b, and consider the appearances of m in the input program spectra p. In the textual similarity computation, the suspiciousness of words are used to determine their weights.

First, we create a vector of weights that represents a bug report and another vector of weights that represents a method. Each element in a vector corresponds to a word that appears in either the bug report or the method. The weight of a word w in document (i.e., bug report or method) d of method corpus C considering program spectra p is:

$$\begin{aligned}\text{SSTFIDF}(w,p,d,C) =& \text{SS}_{\text{word}}(w,p) \times \log(f(w,d)+1) \\ & \times \log \frac{|C|}{|d_i \in C : w \in d_i|}\end{aligned}$$

In the above formula, $\text{SS}_{\text{word}}(w,p)$ is the suspiciousness score of word w computed by Equation 1, $f(w,d)$ is the number of times word w appears in document d, and $d_i \in C$ means document d_i is in the set of document C. Similarly, $w \in d_i$ means word w belongs to document d_i. The above formula considers the weight of a word based on its suspiciousness, and well-known information retrieval metrics: term frequency (i.e., $\log(f(w,d)+1)$) and inverse document frequency (i.e., $\log \frac{|C|}{|d_i \in C : w \in d_i|}$).

After the two vectors of weights of method m and bug report b are computed, we compute the suspiciousness score of the method m by computing the cosine similarity of these two vectors multiplied by a weighting factor. The formula to compute this score is as follows:

$$\text{AML}^{\text{SuspWord}}(b,p,m,C) = \text{SS}_{\text{method}}(m,p) \times \frac{\sum\limits_{w \in b \cap m} \text{SSTFIDF}(w,p,b,C) \times \text{SSTFIDF}(w,p,m,C)}{\sqrt{\sum\limits_{w \in b} \text{SSTFIDF}(w,p,b,C)^2} \times \sqrt{\sum\limits_{w \in m} \text{SSTFIDF}(w,p,m,C)^2}} \quad (2)$$

Here we use $\text{SS}_{\text{method}}(m,p)$ that computes the suspiciousness score of method m considering program spectra p as the weighting factor. This can be computed by various spectrum-based bug localization tools. By default, we use the same fault localization tool as the one used in AML$^{\text{Spectra}}$ component. With this, AML$^{\text{SuspWord}}$ integrates both macro view of method suspiciousness (which considers direct execution of a method in the failing and correct execution traces) and micro view of method suspiciousness (which considers the executions of its constituent words in the execution traces).

3.2 Adaptive Multi-modal Bug Localization

The integrator component serves to combine the scores produced by the three components AML$^{\text{Text}}$, AML$^{\text{Spectra}}$ and AML$^{\text{SuspWord}}$ by taking a weighted sum of the scores. The final suspiciousness score of method m given bug report b and program spectra p in a corpus C is given by:

$$\begin{aligned}f(x_i,\theta) = & \alpha \times \text{AML}^{\text{Text}}(b,m) + \beta \times \text{AML}^{\text{Spectra}}(p,m) \\ & + \gamma \times \text{AML}^{\text{SuspWord}}(b,p,m) \quad (3)\end{aligned}$$

where i refers to a specific (b,p,m) combination (aka *data instance*), x_i denotes the feature vector $x_i = [\text{AML}^{\text{Text}}(b,m), \text{AML}^{\text{Spectra}}(p,m), \text{AML}^{\text{SuspWord}}(b,p,m)]$, and θ is the parameter vector $[\alpha,\beta,\gamma]$, where α,β,γ are arbitrary real numbers. Note that we exclude mentioning corpus C in both sides of Equation 3 to simplify the set of notations used in this section.

The weight parameters (θ) are *tuned adaptively* for a new bug report b based on a set of top-K historical fixed bugs in a training data that are the most similar to b. We find these top-K nearest neighbors by measuring the textual similarity of b with training (historical) bug reports using the VSM model. In this work, we propose a probabilistic learning approach which analyzes this training data to fine-tune the weight parameters α, β, and γ for the new bug report b.

3.2.0.1 Probabilistic Formulation: From a machine learning standpoint, bug localization can be interpreted as a *binary classification* task. For a given combination (b, p, m), the positive label refers to the case when method m is indeed where the bug b is located (i.e., faulty case), and the negative label is when m is not relevant to b (i.e., non-faulty case). As we deal with binary classification task, it is plausible to assume that a data instance follows Bernoulli distribution, c.f., [38]:

$$p(x_i, y_i|\theta) = \sigma(f(x_i, \theta))^{y_i} \left(1 - \sigma(f(x_i, \theta))\right)^{(1-y_i)} \quad (4)$$

where $y_i = 1$ ($y_i = 0$) denotes the positive (negative) label, and $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the logistic function. Using this notation, we can formulate the overall data *likelihood* as:

$$p(X, y|\theta) = \prod_{i=1}^{N} \sigma(f(x_i, \theta))^{y_i} \left(1 - \sigma(f(x_i, \theta))\right)^{(1-y_i)} \quad (5)$$

where N is the total number of data instances (i.e., (b, p, m) combinations), and $y = [y_1, \ldots, y_i, \ldots, y_N]$ is the label vector.

Our primary interest here is to infer the *posterior* probability $p(\theta|X)$, which can be computed via the Bayes' rule:

$$p(\theta|X, y) = \frac{p(X, y|\theta)p(\theta)}{p(X, y)} \quad (6)$$

Specifically, our goal is to find an optimal parameter vector θ^* that maximizes the posterior $p(\theta|X, y)$. This leads to the following optimization task:

$$\begin{aligned} \theta^* &= \arg\max_{\theta} p(\theta|X, y) \\ &= \arg\max_{\theta} p(X, y|\theta)p(\theta) \\ &= \arg\min_{\theta} \left(-\log(p(X, y|\theta)) - \log(p(\theta))\right) \end{aligned} \quad (7)$$

Here we can drop the denominator $p(X, y)$, since it is independent of the parameters θ. The term $p(\theta)$ refers to the *prior*, which we define to be a Gaussian distribution with (identical) zero mean and inverse variance λ:

$$p(\theta) = \prod_{j=1}^{J} \sqrt{\frac{\lambda}{2\pi}} \exp\left(-\frac{\lambda}{2}\theta_j^2\right) \quad (8)$$

where the number of parameters J is 3 in our case (i.e., α, β, and γ).

By substituting (5) and (8) into (7), and by droppping the constant terms that are independent of θ, the optimal parameters θ^* can be computed as:

$$\theta^* = \arg\min_{\theta} \left(\sum_{i=1}^{N} \mathcal{L}_i + \frac{\lambda}{2}\sum_{j=1}^{J} \theta_j^2\right) \quad (9)$$

where $\mathcal{L}_i$ is called the instance-wise loss, as given by:

$$\mathcal{L}_i = -\left[y_i \log(\sigma(f(x_i, \theta))) + (1 - y_i)\log(1 - \sigma(f(x_i, \theta)))\right] \quad (10)$$

Solution to this minimization task is known as the *regularized logistic regression*. The regularization term $\frac{\lambda}{2}\sum_{j=1}^{J} \theta_j^2$– which stems from the prior $p(\theta)$–serves to penalize large parameter values, thereby reducing the risk of data overfitting.

3.2.0.2 Algorithm: To estimate θ^*, we develop an iterative parameter tuning strategy that performs a descent move along the negative gradient of $\mathcal{L}_i$. Algorithm 1 summarizes our proposed parameter tuning method. More specifically, for each instance i, we perform gradient descent update for each parameter θ_j:

$$\theta_j \leftarrow \theta_j - \eta\left(\frac{\partial \mathcal{L}_i}{\partial \theta_j} + \lambda\theta_j\right) \quad (11)$$

where the gradient term $\frac{\partial \mathcal{L}_i}{\partial \theta_j}$ resolves to:

$$\frac{\partial \mathcal{L}_i}{\partial \theta_j} = \left(\sigma(f(x_i, \theta)) - y_i\right) x_{i,j} \quad (12)$$

with the feature values $x_{i,1} = \text{AML}^{\text{Text}}(b, m)$, $x_{i,2} = \text{AML}^{\text{Spectra}}(p, m)$ and $x_{i,3} = \text{AML}^{\text{SuspWord}}(b, p, m)$, corresponding to the parameters α, β and γ, respectively. The update steps are realized in lines 11-13 of Algorithm 1.

One key challenge in the current bug localization task is the extremely skewed distribution of the labels, i.e., the number of positive cases is much smaller than the number of negative cases. To address this, we devise a *balanced random sampling* procedure when picking a data instance for gradient descent update. In particular, for every update step, we alternatingly select a random instance from the positive and negative instance pools, as per lines 4-8 of Algorithm 1.

Using this simple method, we can balance the training from positive and negative instances, thus effectively mitigating the issue of *skewed distribution* in the localization task. It is also worth noting that our iterative tuning procedure is efficient. That is, its time complexity is linear with respect to the number of instances N and maximum iterations T_{max}.

3.3 Generalized Adaptive Multi-modal Bug Localization

The original Adaptive Multi-modal bug Localization (AML) (mentioned in Section 3.2) remains two main issues. Firstly, for each bug report b, a fixed parameter vector $\vec{\theta_b} = (\alpha, \beta, \gamma)$ is inferred for all methods m (see Equation 3). In other words, every method m shares the same parameter vector $\vec{\theta_b}$ when estimating the suspiciousness scores of m according to bug report b and program spectra p. In particular, each method m not only considers the parameter of bug report b but also share the parameter of method m. Therefore, Equation 3 does not accurately reflect the suspiciousness score of method m given bug report b and program spectra p. Secondly, the parameter vector $\vec{\theta_b}$ of a bug report is learned independently of other bug report b', without exploiting the clustering and/or similarity properties of different bug reports and methods. In practice, some bug reports as well as methods are similar to other bug reports/methods. However, the instance-wise loss function in equation 10 ignores

Algorithm 1 Adaptive multi-modal bug localization

```
Require: Matrix $X \in \mathbb{R}^{N\times 3}$ (each row is a vector $x_i = [\text{AML}^{\text{Text}}(b,m), \text{AML}^{\text{Spectra}}(p,m), \text{AML}^{\text{SuspWord}}(b,p,m)]$ for bug report $b$, program spectra $p$, and method $m$ in one of the top-K most similar training data), label vector $y \in \mathbb{R}^N$ (each element $y_i$ is the label of $x_i$), learning rate $\eta$, regularization parameter $\lambda$, maximum training iterations $T_{max}$
Ensure: Weight parameters $\alpha$, $\beta$, $\gamma$
1: Initialize $\alpha$, $\beta$, $\gamma$ to zero
2: repeat
3:     for each $n \in \{1,\dots,N\}$ do
4:         if $n \bmod 2 = 0$ then    ▷ Draw a positive instance
5:             Randomly pick $i$ from $\{1,\dots,N\}$ s.t. $y_i = 1$
6:         else    ▷ Draw a negative instance
7:             Randomly pick $i$ from $\{1,\dots,N\}$ s.t. $y_i = 0$
8:         end if
9:         Compute overall score $f(x_i,\theta)$ using Eq. (3)
10:        Compute gradient $g_i \leftarrow \sigma(f(x_i,\theta)) - y_i$
11:        $\alpha \leftarrow \alpha - \eta\left(g_i \times \text{AML}^{\text{Text}}(b,m) + \lambda\alpha\right)$
12:        $\beta \leftarrow \beta - \eta\left(g_i \times \text{AML}^{\text{Spectra}}(p,m) + \lambda\beta\right)$
13:        $\gamma \leftarrow \gamma - \eta\left(g_i \times \text{AML}^{\text{SuspWord}}(b,p,m) + \lambda\gamma\right)$
14:    end for
15: until $T_{max}$ iterations
```

the relationship between the bug reports and the methods. In light of these issues, we proposed a new algorithm, namely generalized adaptive multi-modal bug localization, to tackle these problems in bug localization.

Our proposed algorithm is inspired by network lasso [19]. In the next sub section, I briefly present a short introduction about network lasso, then explain the proposed algorithm, i.e. Generalized Adaptive Multi-modal bug Localization (AML*).

3.3.1 Network Lasso

Network lasso [19] is a generalization of the group lasso to a network setting that allows for simultaneous clustering and optimization on graphs. In [19], Hallac et al. focused on optimization problem posed on graph. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the vertex set and $\mathcal{E}$ is the edges of graph. The network lasso problem is defined as following:

$$minimize \sum_{i\in\mathcal{V}} f_i(x_i) + \lambda \sum_{(j,k)\in\mathcal{E}} w_{jk}\|x_j - x_k\|_2 \tag{13}$$

The variable are $x_1,\dots,x_n \in R^p$, where $n = |\mathcal{V}|$. $x_i \in R^p$ is the variable at node i, f_i is the cost function at node i, w_{jk} is the relative weights among the edges (j,k) of the network $\mathcal{G}$, and λ is an overall parameter that scales the edge objectives relative to the node objectives. In order to solve the problem 13 in a distributed and scalable manner, which allows for guaranteed global convergence even on large graphs, the author based on the Alternating Direction Method of Multipliers (ADMM) [39], [13]. With ADMM, each individual component solves its own private objective function, passes this solution to its neighbors, and repeats the process until the entire network converges.

3.3.2 Extension Framework

We propose Generalized Adaptive Multi-modal bug Localization (AML*) that involves two extensions. The first is to redefine the function of suspiciousness score (see equation 3) as an interaction between a bug reports parameter vector u_b and a methods parameter vector v_m, as follows:

$$\begin{aligned} f(x_i, u_b, v_m) &= \sum_{j=1}^{J}(u_{b,j} + v_{m,j})x_{i,j} \\ &= \sum_{j=1}^{J} u_{b,j}x_{i,j} + \sum_{j=1}^{J} v_{m,j}x_{i,j} \end{aligned} \tag{14}$$

where x_i is denoted as feature vector of specific (b,p,m). J is a number of features, hence $J = 3$ in our model since we only have three components AML$^{\text{Text}}$, AML$^{\text{Spectra}}$ and AML$^{\text{SuspWord}}$ (see section 3.2). Different to equation 3, each feature vector (b,p,m) not only considers the parameter of bug report b but also share the parameter of method m.

The second extension is the use of the network lasso [19] regularization that exploits the similarities amongst different bug reports as well as methods. Different from the original network lasso that only deals with a single graph, our approach involves optimizing over two graphs simultaneously. Specifically, we consider a joint optimization over the bug similarity graph $\mathcal{G}^B = \{e_{b,b'} | (b,b') \in B\}$, and method similarity graph $\mathcal{G}^M = \{e_{m,m'} | (m,m') \in M\}$.

With the new model parameterization and similarity graphs taken into account, we propose a new dual network lasso-regularized loss function as following:

$$\begin{aligned} \mathcal{L}_i = &-\sum_{b=1}^{B}\sum_{m=1}^{M} w_i \Big[y_i log(\sigma(f(x_i,u_b,v_m))) \\ &+ (1-y_i)log(1-\sigma(f(x_i,u_b,v_m)))\Big] \\ &+ \frac{\alpha}{2}\sum_{j=1}^{J}\left[\sum_{b=1}^{n_B} u_{b,j}^2 + \sum_{m=1}^{n_M} v_{m,j}^2\right] \\ &+ \frac{\beta}{2}\sum_{j=1}^{J}\Big[\sum_{(b,b'\in\mathcal{G}^B)} e_{b,b'}(u_{b,j} - u_{b',j})^2 \\ &+ \sum_{(m,m'\in\mathcal{G}^M)} e_{m,m'}(u_{m,j} - u_{m',j})^2\Big] \end{aligned} \tag{15}$$

where B and M are the number of bug reports and methods in our dataset, u_b and v_m are called the parameter vector of bug reports and methods respectively, y_i denotes the label of method m given bug report b, $\sigma(x_i)$ is the logistic function [15], w_i used to automatically adjust weights inversely proportional to class frequencies in the training data. In AML*, w_i is applied to solve the skewed distribution problem in bug localization task, i.e. the number of positive labels is much smaller than the number of negative labels. α and β are the learning rate of regularization in our loss function. Intuitively, the implication of the network lasso regularization is straightforwardthe more similar two bug reports (or two methods) are, the closer their parameter vectors (i.e., u_b and v_m) should be.

To minimize $\mathcal{L}_i$, we employ a Newton method [23] stemming from a second-order Taylor series expansion of the loss function $\mathcal{L}_i$, which are defined as following:

$$\mathcal{L}_i(\theta) = \mathcal{L}_i(\theta_0) + \triangledown\mathcal{L}_i(\theta_0)(\theta - \theta_0) + \frac{\triangledown^2\mathcal{L}(\theta_0)}{2}(\theta - \theta_0)^2 \quad (16)$$

The minimal of $\mathcal{L}_i$ can be obtained by taking the (partial) derivative of $\mathcal{L}(\theta)$ and equating it to zero:

$$\begin{aligned} 0 &= \triangledown\mathcal{L}_i(\theta_0) + \triangledown^2\mathcal{L}_i(\theta_0)(\theta - \theta_0) \\ \theta &= \theta_0 - \frac{\triangledown\mathcal{L}_i(\theta_0)}{\triangledown^2\mathcal{L}_i(\theta_0)} \end{aligned} \quad (17)$$

If we take θ_0 as the old estimate of $u_{b,j}$ or $v_{m,j}$, this leads to the following update procedures:

$$u_{b,j} \leftarrow u_{b,j} - \frac{\triangledown\mathcal{L}_i(u_{b,j})}{\triangledown^2\mathcal{L}_i(u_{b,j})} \quad (18)$$

$$v_{m,j} \leftarrow v_{m,j} - \frac{\triangledown\mathcal{L}_i(v_{m,j})}{\triangledown^2\mathcal{L}_i(v_{m,j})} \quad (19)$$

In turn, we need to compute the first and second derivatives of each parameter $u_{b,j}$ and $v_{m,j}$. For the bug report parameter $u_{b,j}$, the first and second derivatives are respectively:

$$\begin{aligned} \triangledown\mathcal{L}_i(u_{b,j}) = &\sum_{m=1}^{n_M} \left[w_i(\sigma(f(x_i, u_b, v_m)) - y_i)x_{i,j}\right] \\ &+ \alpha u_{b,j} + \beta \sum_{b'} \left[e_{b,b'}\left(u_{b,j} - u_{b',j}\right)\right] \end{aligned} \quad (20)$$

$$\begin{aligned} \triangledown^2\mathcal{L}_i(u_{b,j}) = &\sum_{m=1}^{n_M} \left[w_i\sigma(f(x_i, u_b, v_m))(1 - \sigma_i)x_{i,j}^2\right] \\ &+ \alpha + \beta \sum_{b'} e_{b,b'} \end{aligned} \quad (21)$$

Similarly, we can compute the first and second derivatives w.r.t each method parameter $v_{m,j}$ as:

$$\begin{aligned} \triangledown\mathcal{L}_i(v_{m,j}) = &\sum_{b=1}^{n_B} \left[w_i(\sigma(f(x_i, u_b, v_m)) - y_i)x_{i,j}\right] \\ &+ \alpha v_{m,j} + \beta \sum_{m'} \left[e_{m,m'}\left(v_{m,j} - v_{m',j}\right)\right] \end{aligned} \quad (22)$$

$$\begin{aligned} \triangledown^2\mathcal{L}_i(v_{m,j}) = &\sum_{b=1}^{n_B} \left[w_{b,m}\sigma_{b,m}(1 - \sigma_{b,m})x_{i,j}^2\right] \\ &+ \alpha + \beta \sum_{m'} e_{m,m'} \end{aligned} \quad (23)$$

Finally, the update formula for $u_{b,j}$ can be obtained by substituting equation 20 and 21 into equation 18. Likewise, by substituting equation 22 and 23 into 19 to update formula $v_{m,j}$. To learn the model parameters, we use a *Newton method* that updates the parameters on a per-feature j basis.

Algorithm 2 summarizes our learning procedure. Intuitively, AML* includes two basic steps: i.e., we first update bug report parameter vector (see step 8 of algorithm 2) and then update method parameter vector (in step 17). For computational efficiency, we precompute the constant terms $q_b = \sum_{b'} e_{b,b'}$ and $q_m = \sum_{m'} e_{m,m'}$ before the Newton iterations begins. During each Newton iteration, we also precompute the terms $\sum_{b'} e_{b,b'}u_{b',j}$ and $\sum_{m'} e_{m,m'}v_{m',j}$ for each feature j.

Algorithm 2 Parameters learning for generalized adaptive multi-modal bug localization

```
Require: Matrix $X \in \mathbb{R}^{B \times M \times 3}$, where $B$ and $M$ are the number of bug reports and methods in our training data. Each cell in 3-D matrix is a vector $x_{b,m} = [\text{AML}^{\text{Text}}(b,m), \text{AML}^{\text{Spectra}}(p,m), \text{AML}^{\text{SuspWord}}(b,p,m)]$ for bug report $b$, program spectra $p$, and method $m$ in one of the top-K most similar training data), label vector $y \in \mathbb{R}^{B \times M}$ (each element $y_{b,m}$ is the label of $x_{b,m}$), sample weight $w \in \mathbb{R}^{B \times M}$, learning rate $\eta$, regularization parameter $\alpha$ and $\beta$, maximum training iterations $T_{max}$
Ensure: Weight parameters $U \in \mathbb{R}^{B \times 3}$ and $\in \mathbb{R}^{M \times 3}$
 1: Initialize all parameters $u_{b,j} \leftarrow 0$ and $v_{m,j} \leftarrow 0$
 2: Precompute constant terms $q_b \leftarrow \sum_{b'} e_{b,b'}$ and $q_m \leftarrow \sum_{m'} e_{m,m'}$
 3: Set the initial learning rate $\eta \leftarrow 1$
 4: repeat
 5:   Compute all predictions $\sigma(f(x_{b,m}, u_b, v_m))$ (see equation 14)
 6:   $\mathcal{L}_{prev} \leftarrow -\sum_b \sum_m w_{b,m}\big[y_{b,m}\log\left(\sigma(f(x_{b,m}, u_b, v_m))\right) + (1 - y_{b,m})\log\left(1 - \sigma(f(x_{b,m}, u_b, v_m))\right)\big]$
 7:   for each $j \in \{1, \ldots, 3\}$ do
 8:     /* Update all $u_{b,j}$ */
 9:     for each $b \in \{1, \ldots, B\}$ do
10:       $p_b \leftarrow \sum_{b'} e_{b,b'}u_{b',j}$
11:     end for
12:     for each $b \in \{1, \ldots, B\}$ do
13:       $b_{num} \leftarrow \sum_{m=1}^{M}\left[w_{b,m}(\sigma(f(x_{b,m}, u_b, v_m)) - y_{b,m})x_{b,m,j}\right] + \beta\left[u_{b,j}q_b - p_b\right] + \alpha u_{b,j}$
14:       $b_{deno} \leftarrow \sum_{m=1}^{M}\left[w_{b,m}\sigma(f(x_{b,m}, u_b, v_m))(1 - \sigma(f(x_{b,m}, u_b, v_m))x_{b,m,j}^2\right] + \beta q_b + \alpha$
15:       $u_{b,j} \leftarrow u_{b,j} - \eta\left(\frac{b_{num}}{b_{deno}}\right)$
16:     end for
17:     /* Update all $v_{m,j}$ */
18:     for each $m \in \{1, \ldots, M\}$ do
19:       $p_m \leftarrow \sum_{m'} e_{m,m'}v_{m',j}$
20:     end for
21:     for each $m \in \{1, \ldots, M\}$ do
22:       $v_{num} \leftarrow \sum_{b=1}^{B}\left[w_{b,m}(\sigma(f(x_{b,m}, u_b, v_m)) - y_{b,m})x_{b,m,j}\right] + \beta\left[v_{m,j}q_m - p_m\right] + \alpha v_{m,j}$
23:       $v_{deno} \leftarrow \sum_{b=1}^{B}\left[w_{b,m}\sigma(f(x_{b,m}, u_b, v_m))(1 - \sigma(f(x_{b,m}, u_b, v_m)))x_{b,m,j}^2\right] + \beta q_m + \alpha$
24:       $v_{m,j} \leftarrow v_{m,j} - \eta\left(\frac{v_{num}}{v_{deno}}\right)$
25:     end for
26:   end for
27:   Compute all predictions $\sigma(f(x_{b,m}, u_b, v_m)$
28:   $\mathcal{L}_{curr} \leftarrow -\sum_b \sum_m w_{b,m}\big[y_{b,m}\log\left(\sigma(f(x_{b,m}, u_b, v_m))\right) + (1 - y_{b,m})\log\left(1 - \sigma(f(x_{b,m}, u_b, v_m))\right)\big]$
29:   $\eta \leftarrow \begin{cases} \frac{\eta}{2}, & \text{if } \mathcal{L}_{curr} > \mathcal{L}_{prev} \\ \min(1, 2\eta), & \text{otherwise} \end{cases}$
30: until $T_{max}$ iterations
```

4 Experiments

4.1 Dataset

We use a dataset of 157 bugs from 4 popular software projects to evaluate our approach against the baselines.

TABLE 3
Dataset Description

Project	#Bugs	Time Period	Average # Methods
AspectJ	41	03/2005 – 02/2007	14,218.39
Ant	53	12/2001 – 09/2013	9,624.66
Lucene	37	06/2006 – 01/2011	10,220.14
Rhino	26	12/2007 – 12/2011	4,839.58

These projects are AspectJ [3], Ant [1], Lucene [2], and Rhino [7]. All four projects are medium-large scale and implemented in Java. AspectJ, Ant, and Lucene contain more than 300 kLOC, while Rhino contains almost 100 kLOC. Table 3 describes detailed information of the four projects in our study.

The 41 AspectJ bugs are from the iBugs dataset which were collected by Dallmeier and Zimmermann [16]. Each bug in the iBugs dataset comes with the code before the fix (pre-fix version), the code after the fix (post-fix version), and a set of test cases. The iBugs dataset contains more than 41 AspectJ bugs but not all of them come with failing test cases. Test cases provided in the iBugs dataset are obtained from the various versions of the regression test suite that comes with AspectJ. The remaining 116 bugs from Ant, Lucene, and Rhino are collected by ourselves following the procedure used by Dallmeier and Zimmermann [16]. For each bug, we collected the pre-fix version, post-fix version, a set of successful test cases, and at least one failing test case. A failing test case is often included as an attachment to a bug report or committed along with the fix in the post-fix version. When a developer receives a bug report, he/she first needs to replicate the error described in the report [37]. In this process, he is creating a failing test case. Unfortunately, not all test cases are documented and saved in the version control systems.

4.2 Evaluation Metric and Settings

We use two metrics namely mean average precision (MAP) and Top N to evaluate the effectiveness of a bug localization solution. They are defined as follows:

- **Top N**: Given a bug, if one of its faulty methods is in the top-N results, we consider the bug is successfully localized. Top N score of a bug localization tool is the number of bugs that the tool can successfully localize [65], [47].
- **Mean Average Precision (MAP)**: MAP is an IR metric to evaluate ranking approaches [35]. MAP is computed by taking the mean of the *average precision* scores across all bugs. The average precision of a single bug is computed as:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{number\ of\ buggy\ methods}$$

where k is a rank in the returned ranked methods, M is the number of ranked methods, and $pos(k)$ indicates whether the k^{th} method is faulty or not. $P(k)$ is the precision at a given top k methods and is computed as follows:

$$P(k) = \frac{\#faulty\ methods\ in\ the\ top\ k}{k}.$$

Note that typical MAP scores of existing bug localization techniques are low [44], [50], [65], [47].

We use 10 fold cross validation: for each project, we divide the bugs into ten sets, and use 9 as training data and 1 as testing data. We repeat the process 10 times using different training and testing data combinations. We then aggregate the results to get the final Top N and MAP scores.

In Network-regularized Bug Localization (NetBL), the initial learning rate η is set to one, and decreases $\frac{\eta}{2}$ if the current value of loss function is larger than previous one to speed up the learning process (see step 29 in algorithm 2). The regularization parameter α and β are chosen by performing 10 fold cross validation on the training data. The maximum number of iterations T_{max} is fixed as 30. We use $K = 10$ as default value for the number of nearest neighbors. Note that the NetBL parameters (i.e. η, K and T_{max}) are chosen similar to [25]. We conduct experiments on an Intel(R) Xeon E5-2667 2.9GHz server running Linux 2.6.

We first compare NetBL with original multi-modal techniques (i.e. Adaptive Multi-Modal bug Localization)

We compare the two approaches (i.e., AML* and AML) against 3 state-of-the-art multi-modal feature localization techniques (i.e., PROMESIR [42], DIT^A and DIT^B [18]), a state-of-the-art IR-based bug localization technique named LR [61], and a state-of-the-art spectrum-based bug localization technique named MULTRIC [60]. We use the same parameters and settings that are described in their papers with the following exceptions that we justify. For DIT^A and DIT^B, the threshold used to filter methods using HITS was decided "such that at least one gold set method remained in the results for 66% of the [bugs]" [18]. In this paper, since we use ten-fold cross validation, rather than using 66% of all bugs, we use all bugs in the training data (i.e., 90% of all bugs) to tune the threshold. For PROMESIR, we also use 10-fold CV and apply a brute force approach to tune PROMESIR's component weights using a step of 0.05. PROMESIR, DIT^A, DIT^B, and MULTRIC locate buggy methods, however LR locate buggy files. Thus, we convert the list of files that LR produces into a list of methods by using two heuristics: (1) return methods in a file in the same order that they appear in the file; (2) return methods based on their similarity to the input bug report as computed using a VSM model. We refer to the two variants of LR as LR^A and LR^B respectively.

4.3 Research Questions

Research Question 1: How effective are AML* and AML as compared to state-of-the-art techniques?

PROMESIR [42], SITIR [30], and several algorithm variants proposed by Dit et al. [18] are state-of-the-art multi-modal feature location techniques. Among the variants proposed by Dit et al. [18], the best performing ones are $IR_{LSI}Dyn_{bin}WM_{HITS}(h, bin)^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS}(h, freq)^{bottom}$. We refer to them as DIT$^{\mathrm{A}}$ and DIT$^{\mathrm{B}}$ in this paper. Dit et al. have shown that these two variants outperform SITIR. However, Dit et al.'s variants have never been compared with PROMESIR. PROMESIR has also never been compared with SITIR. Thus, to answer this research question, we compare the performance of our approach with PROMESIR, DIT$^{\mathrm{A}}$ and DIT$^{\mathrm{B}}$. We also compare with the two variants of LR [61] (LR^{A} and LR^{B}) and MULTRIC [60] which are recently proposed state-of-the-art IR-based and spectrum-based bug localization techniques respectively.

Research Question 2: Are the results of AML* and AML significant?

We use Wilcoxon signed-rank test [52] with significant level 0.05 to compare the results of AML* and AML. More specifically, we would like to know whether the results of AML* and AML are significant. The Wilcoxon signed-rank test are applied for two different evaluation metrics: TopN and MAP (see section 4.2). For each metric, we employ Wilcoxon signed-rank to each software project (i.e., AspectJ, Ant, Lucene, and Rhino) to see whether there are any significant results of AML* and AML. Moreover, the statistical test are also employed across all software projects without considering project domain. Our null hypothesis is the results AML* outperforms AML, and hence we apply one-tailed Wilcoxon signed-rank test.

Research Question 3: Are all components of AML* and AML contributing toward its overall performance?

To answer this research question, we simply drop one feature component (i.e., text, suspword, and spectra) from all components one-at-a-time and evaluate their performance. In the process, we create three variants of proposed approaches: All$^{-\mathrm{Text}}$, All$^{-\mathrm{SuspWord}}$, and All$^{-\mathrm{Spectra}}$. In particular, we exclude text, suspword, and spectra components (see figure 2). We use the default value of $K = 10$, and apply AML* and AML to tune weights of these variants, and compare their performance with our methods.

Research Question 4: How effective are our proposed approaches component (i.e. AML* and AML)?

Rather than using the our proposed component, it is possible to use a standard machine learning algorithm, e.g., learning-to-rank, to combine the scores produced by feature components (i.e., text, suspword, and spectra). Indeed, the two state-of-the-art IR-based and spectrum-based bug localization techniques (i.e., LR and MULTRIC) are based on learning-to-rank. In this research question, we want to compare our Integrator component with an off-the-shelf learning-to-rank tool namely SVM^{rank} [21], which was also used by LR [61]. We simply replace the Integrator component with SVM^{rank} and evaluate the effectiveness of the resulting solution.

Research Question 5: What is the effect of varying the number of neighbors K on the performance of AML* and AML?

Our proposed approach takes as input one parameter, which is the number of neighbors K, that is used to adaptively tune the weights α, β, and γ for a bug. By default, we set the number of neighbors to 10. The effect of varying this default value is unclear. To answer this research question, we vary the value of K and we investigate the effect of different numbers of neighbors on the performance of AML* and AML. In particular, we want to investigate if the performance of AML* remains relatively stable for a wide range of K.

4.4 Results

4.4.1 *RQ1: Proposed approaches vs. Baselines*

Table 4 shows the performance of AML*, AML, and all the baselines in terms of Top N. Out of the 157 bugs, AML* can successfully localize 46, 82, and 100 bugs when developers inspect the top 1, top 5, and top 10 methods respectively. This means that AML* can successfully localize 119.05%, 51.85%, and 38.89% more bugs than the best baseline (i.e., PROMESIR) by investigating the top 1, top 5, and top 10 methods respectively. Moreover, table 4 also shows that AML* outperforms 48.39%, 15.49%, and 8.7% in the top 1, top 5, and top 10 methods compared to AML.

Table 5 shows the performance of AML*, AML and the baselines in terms of MAP. AML* achieves MAP scores of 0.219, 0.270, 0.290, and 0.302 for AspectJ, Ant, Lucene, and Rhino datasets, respectively. Averaging across the four projects, both AML* and AML achieve an overall MAP score of 0.270 and 0.237 respectively which outperform all the baselines (i.e. PROMESIR, DIT, LR and MULTRIC). In particular, AML* beats the average MAP scores of PROMESIR, DIT$^{\mathrm{A}}$, DIT$^{\mathrm{B}}$, LR^{A}, LR^{B}, and MULTRIC by 46.74%, 128.81%, 141.07%, 527.91%, 112.60%, and 138.94%, whereas AML improves the average MAP scores of AML, PROMESIR, DIT$^{\mathrm{A}}$, DIT$^{\mathrm{B}}$, LR^{A}, LR^{B}, and MULTRIC by 28.80%, 100.85%, 111.61%, 451.16%, 91.34%, and 109.73% respectively. We also see that AML* surpasses the overall MAP scores of AML by 13.92%.

Considering each individual project, in terms of MAP, AML* and AML are still the best performing multi-modal bug localization approach. AML* beats the MAP score of the best performing baseline (i.e. PROMESIR), by 80.99%, 31.07%, 42.16%, and 48.77% for AspectJ, Ant, Lucene, and Rhino datasets, respectively. While AML outperforms the MAP score of PROMESIR, by 54.55%, 13.59%, 39.22%, and 19.70% for AspectJ, Ant, Lucene, and Rhino datasets, respectively. Moreover, AML* also outperforms the MAP score of AML by 17.11%, 15.38%, 2.11% and 24.28% across four different datasets (i.e. AspectJ, Ant, Lucene, and Rhino).

4.4.2 *RQ2: Significant results of AML* and AML*

Table 4 shows the results of Wilcoxon signed-rank test of AML* and AML among Top N. $(*)$ means that results of AML* and AML are significant. AML* and AML are statistically significant in terms of top 1 methods among AspectJ, Ant, and Rhino at significant level 0.05 of Wilcoxon signed-rank test. Moreover, we see that AML* and AML show significant results among all the four software projects in the top 1 methods. Among top 5 methods, only AML* and AML are only significant in project Rhino. In the top 10 methods, AML* and AML do not show any significant results in all software projects.

Table 5 presents the Wilcoxon signed-rank test of AML* and AML in terms of MAP. Similar to top 1 methods, AML* and AML are significant in AspectJ, Ant and Rhino, however they are not significant in project Lucene. However, both methods are significant across four different software data.

TABLE 4
Top N: AML* and AML vs. Baselines. $N \in \{1, 5, 10\}$. P = PROMESIR, D = DIT, L = LR, and M = MULTRIC.

Top	Project	AML*	AML	P	D^A	D^B	L^A	L^B	M
1	AspectJ	11	7*	4	4	3	0	0	0
	Ant	13	9*	7	3	3	1	11	2
	Lucene	12	11	8	7	7	1	7	4
	Rhino	10	4*	2	1	1	0	2	2
	Overall	**46**	31*	21	15	14	2	20	8
5	AspectJ	15	13	6	4	3	0	0	1
	Ant	24	22	17	10	10	11	20	7
	Lucene	25	22	18	13	13	6	16	13
	Rhino	18	14*	13	5	5	2	8	8
	Overall	**82**	71	54	32	31	19	44	29
10	AspectJ	16	13	9	4	3	0	0	2
	Ant	35	31	28	20	20	19	32	15
	Lucene	30	29	21	20	19	10	24	16
	Rhino	19	19	14	7	7	3	12	11
	Overall	**100**	92	72	51	49	32	68	44

TABLE 5
Mean Average Precision: AML* and AML vs. Baselines. P = PROMESIR, D = DIT, L = LR, and M = MULTRIC.

Project	AML*	AML	P	D^A	D^B	L^A	L^B	M
AspectJ	0.219	0.187*	0.121	0.092	0.071	0.006	0.004	0.016
Ant	0.270	0.234*	0.206	0.120	0.120	0.070	0.218	0.077
Lucene	0.290	0.284	0.204	0.169	0.166	0.063	0.184	0.188
Rhino	0.302	0.243*	0.203	0.092	0.090	0.034	0.103	0.172
Overall	**0.270**	0.237*	0.184	0.118	0.112	0.043	0.127	0.113

4.4.3 RQ3: Contributions of AML* and AML Components

Table 6 shows the performance of the three variants, and the full of AML* and AML. From the table, the full components of both approaches have the best performance in term of Top 1, Top 5, Top 10, and MAP. This shows that omitting one of the components (i.e., text, suspword, and spectra) reduces the effectiveness of AML* and AML. Thus, *each of the component contributes towards the overall performance of AML* and AML*. Also, among the variants, All$^{-\text{SuspWord}}$ has the smallest Top 1, Top 5, Top 10, and MAP scores in AML* and AML. The reduction in the evaluation metric scores are the largest when we omit All$^{\text{SuspWord}}$. This indicates that the supsword component *is more important than the other components (i.e., text and spectra).*

4.4.4 RQ4: AML* and AML vs. SVM^{rank}

Table 7 shows the results of AML* and AML comparing with SVM^{rank}. We can note that for most subject programs and metrics, our approaches outperforms SVM^{rank}. AML approach build a personalized model for each bug and considers the data imbalance phenomenon, whereas AML* takes advantages of sharing parameter vectors of both bug report b and method m. Moreover, AML* also exploits the similarity properties of different bug reports and methods and takes into account the imbalance problem of our data. Therefore, AML* outperforms AML in term of top 1, top2, top 5 and MAP scores across four different projects (i.e., AspectJ, Ant, Lucene, and Rhino).

4.4.5 RQ5: Effect of Varying Number of Neighbors

To answer this research question, we vary the number of neighbors K from 5 to all bugs in the training data (i.e., $K = \infty$) for both AML* and AML. The results with varying numbers of neighbors is shown in Table 8. We can see that, as we increase K, the performance of AML increases until a certain point. When we use a large K, the performance of AML decreases. This suggests that in general including more neighbors can improve performance. However, an overly large number of neighbors may lead to an increased level of noise (i.e., the number of non-representative neighbors), resulting in a degraded performance. The differences in the Top N and MAP scores are small though. In AML*, the performances of K from 5 to 20 do not show significant difference since AML* considers the similarity properties of different bug reports and methods. However, the performance of two evaluation metrics (i.e., topN and MAP) also decreases because of the noise from large number of neighbors.

4.5 Discussion

Number of Failed Test Cases and Its Impact: In our experiments with 157 bugs, most of the bugs come with few failed test cases (average = 2.185). We investigate whether the number of failed test cases impacts the effectiveness of our approach. We compute the differences between the average number of failed test cases for bugs that are successfully localized at top-N positions (N = 1,5,10) and bugs that are not successfully localized. We find that the differences are small (-0.472 to 0.074 test cases). These indicate that the number of test cases do not impact the effectiveness of our approach much and typically 1 to 3 failed test cases are sufficient for our approach to be effective.

Threats to Validity: Threats to internal validity relate to implementation and dataset errors. We have checked our implementations and datasets. However, still there could be errors that we do not notice. Threats to external validity relate to the generalizability of our findings. In this

TABLE 6
Contributions of AML* and AML in each feature components

Approach	Top 1		Top 5		Top 10		MAP	
	AML*	AML	AML*	AML	AML*	AML	AML*	AML
All$^{-\text{Text}}$	**33**	28	**68**	68	**87**	87	**0.212**	0.212
All$^{-\text{SuspWord}}$	**30**	28	**63**	62	**83**	83	**0.211**	0.201
All$^{-\text{Spectra}}$	**35**	26	**73**	63	**87**	84	**0.220**	0.210
All	**46**	31	**82**	71	**100**	92	**0.270**	0.237

TABLE 7
AML* and AML vs. SVM^{rank}.

Metrics	Project	AML*	AML	SVM^{rank}
Top 1	AspectJ	11	7	4
	Ant	13	9	7
	Lucene	12	11	10
	Rhino	10	4	4
	Overall	**46**	31	25
Top 5	AspectJ	15	13	11
	Ant	24	22	24
	Lucene	25	22	23
	Rhino	18	14	13
	Overall	**82**	71	71
Top 10	AspectJ	16	13	14
	Ant	35	31	31
	Lucene	30	29	26
	Rhino	19	19	16
	Overall	**100**	92	87
MAP	AspectJ	0.219	0.187	0.131
	Ant	0.270	0.234	0.234
	Lucene	0.290	0.284	0.267
	Rhino	0.302	0.243	0.227
	Overall	**0.270**	0.237	0.215

work, we have analyzed 157 real bugs from 4 medium-large software systems. In the future, we plan to reduce the threats to external validity by investigating more real bugs from additional software systems, written in various programming languages. Threats to construct validity relate to the suitability of our evaluation metrics and experimental settings. Both Top N and MAP have been used to evaluate many past bug localization studies [44], [50], [65], [47]. MAP is also well known in the information retrieval community [35]. We perform cross validation to evaluate the effectiveness of approach on various training and test data. Cross validation is a standard setting used to evaluate many past studies [10], [20], [17], [49]. Unfortunately, cross validation ignores temporal ordering among bug reports. If bugs reported at different dates do not exhibit substantially different characteristics in terms of their program spectra and descriptions, then this threat is minimal.

5 Related Work

Multi-Modal Feature Location: Multi-modal feature location takes as input a feature description and a program spectra, and finds program elements that implement the corresponding feature. There are several multi-modal feature location techniques proposed in the literature [42], [30], [18].

Poshyvanyk et al. proposed an approach named PROMESIR that computes weighted sums of scores returned by an IR-based feature location solution (LSI [36]) and a spectrum-based solution (Tarantula [22]), and rank program elements based on their corresponding weighted sums [42]. Then, Liu et al. proposed an approach named SITIR which filters program elements returned by an IR-based feature location solution (LSI [36]) if they are not executed in a failing execution trace [30]. Later, Dit et al. used HITS, a popular algorithm that ranks the importance of nodes in a graph, to filter program elements returned by SITIR [18]. Several variants are described in their paper and the best performing ones are $IR_{LSI}Dyn_{bin}WM_{HITS}(h, bin)^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS}(h, freq)^{bottom}$. We refer to these two as DIT^{A} and DIT^{B}, respectively. They have showed that these variants outperform SITIR, though they have never been compared with PROMESIR.

In this work, we compare our proposed approach against PROMESIR, DIT^{A} and DIT^{B}. We show that our approach outperforms all of them on all datasets.

IR-Based Bug Localization: Various IR-based bug localization approaches that employ information retrieval techniques to calculate the similarity between a bug report and a program element (e.g., a method or a source code file) have been proposed [44], [34], [26], [50], [65], [47], [55], [56], [61].

Lukins et al. used a topic modeling algorithm named Latent Dirichlet Allocation (LDA) for bug localization [34]. Then, Rao and Kak evaluated the utility of many standard IR techniques for bug localization including VSM and Smoothed Unigram Model (SUM) [44]. In the IR community, historically, VSM is proposed very early (four decades ago by Salton et al. [48]), followed by many other IR techniques, including SUM and LDA, which address the limitations of VSM.

More recently, a number of approaches which considers information aside from text in bug reports to better locate bugs were proposed. Sisman and Kak proposed a version history aware bug localization technique which considers past buggy files to predict the likelihood of a file to be buggy and uses this likelihood along with VSM to localize bugs [50]. Around the same time, Zhou et al. proposed an approach named BugLocator that includes a specialized VSM (named rVSM) and considers the similarities among bug reports to localize bugs [65]. Next, Saha et al. proposed an approach that takes into account the structure of source code files and bug reports and employs structured retrieval for bug localization, and it performs better than BugLocator [47]. Subsequently, Wang and Lo proposed an approach that integrates the approaches by Sisman and Kak, Zhou et al. and Saha et al. for more effective bug localization [55]. Most recently, Ye et al. proposed an approach named LR that combines multiple ranking features using learning-to-rank to localize bugs, and these features include surface lexical similarity, API-enriched lexical similarity, collaborative fil-

TABLE 8
Effect of Varying Number of Neighbors (K) on AML* and AML

#Neighbors	Top 1		Top 5		Top 10		MAP	
	AML*	AML	AML*	AML	AML*	AML	AML*	AML
$K = 5$	45	29	82	68	102	84	0.272	0.223
$K = 10$	46	31	82	71	100	92	0.270	0.237
$K = 15$	46	30	83	70	101	91	0.272	0.237
$K = 20$	47	29	82	70	101	88	0.269	0.227
$K = 25$	44	29	81	67	101	87	0.264	0.224
$K = \infty$	39	28	75	69	96	86	0.261	0.222

tering, class name similarity, bug fix recency, and bug fix frequency [61].

All these approaches can be used as the AML$^{\text{Text}}$ component of our approach. In this work, we experiment with a basic IR technique namely VSM. Our goal is to show that even with the most basic IR-based bug localization component, we can outperform existing approaches including the state-of-the-art IR-based approach by Ye et al. [61].

Spectrum-Based Bug Localization: Various spectrum-based bug localization approaches have been proposed in the literature [22], [8], [31], [32], [28], [29], [11], [12], [63], [64], [14], [33]. These approaches analyze a program spectra which is a record of program elements that are executed in failed and successful executions, and generate a ranked list of program elements. Many of these approaches propose various formulas that can be used to compute the suspiciousness of a program element given the number of times it appears in failing and successful executions.

Jones and Harrold proposed Tarantula that uses a suspiciousness score formula to rank program elements [22]. Later, Abreu et al. proposed another suspiciousness formula called Ochiai [8], which outperforms Tarantula. Then, Lucia et al. investigated 40 different association measures and highlighted that some of them including Klosgen and Information Gain are promising for spectrum-based bug localization [31], [32]. Recently, Xie et al. conducted a theoretical analysis and found that several families of suspiciousness score formulas outperform other families [58]. Next, Yoo proposed to use genetic programming to generate new suspiciousness score formulas that can perform better than many human designed formulas [62]. Subsequently, Xie et al. theoretically compared the performance of the formulas produced by genetic programming and identified the best performing ones [59]. Most recently, Xuan and Monperrus combined 25 different suspiciousness score formulas into a composite formula using their proposed algorithm named MULTRIC, which performs its task by making use of an off-the-shelf learning-to-rank algorithm named RankBoost [60]. MULTRIC has been shown to outperform the best performing formulas studied by Xie et al. [58] and the best performing formula constructed by genetic programming [62], [59].

Many of the above mentioned approaches that compute suspiciousness scores of program elements can be used in the AML$^{\text{Spectra}}$ component of our proposed approach. In this work, we experiment with a popular spectrum-based fault localization technique namely Tarantula, published a decade ago, which is also used by PROMESIR [42]. Our goal is to show that even with a basic spectrum-based bug localization component, we can outperform existing approaches including the state-of-the-art spectrum-based approach by Xuan and Monperrus [60].

Other Related Studies. There are many studies that compose multiple methods together to achieve better performance. For example, Kocaguneli et al. combined several single software effort estimation models to create more powerful multi-model ensembles [24]. Also, Rahman et al. used static bug-finding to improve the performance of statistical defect prediction and vice versa [43].

6 Conclusion and Future Work

In this paper, we put forward two novels multi-modal bug localization approach, namely Adaptive Multi-modal bug Localization (AML) and Generalized Adaptive Multi-modal bug Localization (AML*). Different from previous multi-modal approaches that are one-size-fits-all, our proposed approaches can adapt itself to better localize each new bug report by tuning various weights learned from a set of training bug reports that are relevant to the new report. We figure out that suspicious words (i.e., words that are associated to a bug) component can be used to find out more bugs in software systems. We have evaluated our proposed approach on 157 real bugs from 4 different datasets. Our experiments highlight that, among the 157 bugs, AML can successfully localize 31, 71, and 92 bugs when developers inspect the top 1, top 5, and top 10 methods, respectively. Compared to the best performing baseline (i.e., PROMESIR), AML can successfully localize 47.62%, 31.48%, and 27.78% more bugs when developers inspect the top 1, top 5, and top 10 methods, respectively. Furthermore, in terms of MAP, AML outperforms the best baseline by 28.80%. AML* is an extension framework of AML takes into account the similarity properties of bug reports and methods, and outperforms AML by 48.39%, 15.49%, and 8.7% in the top 1, top 5, and top 10 methods. Moreover, AML* also outperforms AML by 13.9% in terms of MAP scores.

Dataset. Additional information of the 157 bugs used in the experiments is available at https://bitbucket.org/amlfse/amldata/downloads/amldata.7z.

References

[1] "Apache Ant," http://ant.apache.org/, accessed: 2015-07-15.

[2] "Apache Lucene," http://lucene.apache.org/core/, accessed: 2015-07-15.

[3] "AspectJ," http://eclipse.org/aspectj/, accessed: 2015-07-15.

[4] "Cobertura: A code coverage utility for Java." http://cobertura.github.io/cobertura/, accessed: 2015-07-15.

[5] "Eclipse Java development tools (JDT)," http://www.eclipse.org/jdt/, accessed: 2015-07-15.

[6] "Mysql 5.6 full-text stopwords," http://dev.mysql.com/doc/refman/5.6/en/fulltext-stopwords.html, accessed: 2015-07-15.

[7] “Rhino,” http://developer.mozilla.org/en-US/docs/Rhino, accessed: 2015-07-15.

[8] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, “A practical evaluation of spectrum-based fault localization,” *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2009.06.035

[9] J. Anvik, L. Hiew, and G. C. Murphy, “Coping with an open bug repository,” in *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005*, 2005, pp. 35–39.

[10] ——, “Who should fix this bug?” in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, 2006, pp. 361–370.

[11] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, “Directed test generation for effective fault localization,” in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy*, 2010, pp. 49–60.

[12] ——, “Practical fault localization for dynamic web applications,” in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 265–274. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806840

[13] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1561/2200000016

[14] H. Cleve and A. Zeller, “Locating causes of program failures,” in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, 2005, pp. 342–351.

[15] M. Collins, R. E. Schapire, and Y. Singer, “Logistic regression, adaboost and bregman distances,” *Mach. Learn.*, vol. 48, no. 1-3, pp. 253–285, Sep. 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1013912006537

[16] V. Dallmeier and T. Zimmermann, “Extraction of bug localization benchmarks from history,” in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, 2007, pp. 433–436.

[17] M. D’Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” in *Proceedings of the 7th International Working Conference on Mining Software Repositories*, 2010, pp. 31–41.

[18] B. Dit, M. Revelle, and D. Poshyvanyk, “Integrating information retrieval, execution and link analysis algorithms to improve feature location in software,” *Empirical Software Engineering*, vol. 18, no. 2, pp. 277–309, 2013.

[19] D. Hallac, J. Leskovec, and S. Boyd, “Network lasso: Clustering and optimization in large graphs,” in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’15. New York, NY, USA: ACM, 2015, pp. 387–396. [Online]. Available: http://doi.acm.org/10.1145/2783258.2783313

[20] P. Hooimeijer and W. Weimer, “Modeling bug report quality,” in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta, Georgia, USA*, 2007, pp. 34–43.

[21] T. Joachims, “Svm^{rank}: Support vector machine for ranking,” www.cs.cornell.edu/people/tj/svm_light/svm_rank.html, accessed: 2015-07-15.

[22] J. A. Jones and M. J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, 2005, pp. 273–282.

[23] C. Kelley, *Solving Nonlinear Equations with Newton’s Method*. Society for Industrial and Applied Mathematics, 2003. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9780898718898

[24] E. Kocaguneli, T. Menzies, and J. W. Keung, “On the value of ensemble effort estimation,” *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1403–1416, 2012.

[25] T.-D. B. Le, R. J. Oentaryo, and D. Lo, “Information retrieval and spectrum based bug localization: Better together,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 579–590. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786880

[26] T. D. B. Le, S. Wang, and D. Lo, “Multi-abstraction concern localization,” in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, 2013, pp. 364–367. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2013.48

[27] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, “Bug isolation via remote program sampling,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, 2003, pp. 141–154.

[28] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation,” in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, 2005, pp. 15–26.

[29] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, “SOBER: statistical model-based bug localization,” in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 286–295.

[30] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, “Feature location via information retrieval based filtering of a single scenario execution trace,” in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, 2007, pp. 234–243.

[31] Lucia, D. Lo, L. Jiang, and A. Budi, “Comprehensive evaluation of association measures for fault localization,” in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, 2010, pp. 1–10.

[32] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, “Extended comprehensive study of association measures for fault localization,” *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[33] Lucia, D. Lo, and X. Xia, “Fusion fault localizers,” in *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 127–138.

[34] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, “Bug localization using latent dirichlet allocation,” *Inf. Softw. Technol.*, vol. 52, no. 9, pp. 972–990, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2010.04.002

[35] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[36] A. Marcus and J. I. Maletic, “Recovering documentation-to-source-code traceability links using latent semantic indexing,” in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, 2003, pp. 125–137.

[37] Mozilla, “Bug fields,” https://bugzilla.mozilla.org/page.cgi?id=fields.html, accessed: 2015-03-16.

[38] K. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[39] N. Parikh and S. Boyd, “Proximal algorithms,” *Found. Trends Optim.*, vol. 1, no. 3, pp. 127–239, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1561/2400000003

[40] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada*, 2011, pp. 199–209.

[41] M. F. Porter, “An algorithm for suffix stripping,” *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[42] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval,” *IEEE Trans. Software Eng.*, vol. 33, no. 6, pp. 420–432, 2007.

[43] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, “Comparing static bug finders and statistical prediction,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 424–434.

[44] S. Rao and A. Kak, “Retrieval from software libraries for bug localization: A comparative study of generic and composite text models,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR ’11. New York, NY, USA: ACM, 2011, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985451

[45] S. Rao and A. C. Kak, “Retrieval from software libraries for bug localization: a comparative study of generic and composite text models,” in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, 2011, pp. 43–52.

[46] M. Renieris and S. P. Reiss, “Fault localization with nearest neighbor queries,” in *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, 2003, pp. 30–39.

[47] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, “Improving bug localization using structured information retrieval,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, 2013, pp. 345–355.

[48] G. Salton, A. Wong, and C. S. Yang, “A vector space model for automatic indexing,” *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[49] S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim, “Reducing features to improve code change-based bug prediction,” *IEEE Trans. Software Eng.*, vol. 39, no. 4, pp. 552–569, 2013.

[50] B. Sisman and A. C. Kak, “Incorporating version histories in information retrieval based bug localization,” in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, ser. MSR ’12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 50–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=2664446.2664454

[51] ——, “Incorporating version histories in information retrieval based bug localization,” in *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, 2012, pp. 50–59.

[52] M. D. Smucker, J. Allan, and B. Carterette, “A comparison of statistical significance tests for information retrieval evaluation,” in *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, ser. CIKM ’07. New York, NY, USA: ACM, 2007, pp. 623–632. [Online]. Available: http://doi.acm.org/10.1145/1321440.1321528

[53] G. Tassey, “The economic impacts of inadequate infrastructure for software testing,” National Institute of Standards and Technology, Tech. Rep., 2002.

[54] F. Thung, S. Wang, D. Lo, and L. Jiang, “An empirical study of bugs in machine learning systems,” in *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*, 2012, pp. 271–280.

[55] S. Wang and D. Lo, “History, similar report, and structure: Putting them together for improved bug localization,” in *ICPC*, 2014.

[56] S. Wang, D. Lo, and J. Lawall, “Compositional vector space models for improved bug localization,” in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 171–180.

[57] X. Xia, X. Zhou, D. Lo, and X. Zhao, “An empirical study of bugs in software build systems,” in *2013 13th International Conference on Quality Software, Najing, China, July 29-30, 2013*, 2013, pp. 200–203.

[58] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, “A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization,” *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, p. 31, 2013.

[59] X. Xie, F. Kuo, T. Y. Chen, S. Yoo, and M. Harman, “Provably optimal and human-competitive results in SBSE for spectrum based fault localisation,” in *Search Based Software Engineering - 5th International Symposium, SSBSE 2013, St. Petersburg, Russia, August 24-26, 2013. Proceedings*, 2013, pp. 224–238.

[60] J. Xuan and M. Monperrus, “Learning to combine multiple ranking metrics for fault localization,” in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 191–200.

[61] X. Ye, R. C. Bunescu, and C. Liu, “Learning to rank relevant files for bug reports using domain knowledge,” in *Proceedings of the 22nd ACM SIGSOFT*

International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, 2014, pp. 689–699.

[62] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Search Based Software Engineering - 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings*, 2012, pp. 244–258.

[63] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transaction on Software Engineering*, vol. 28, pp. 183–200, 2002.

[64] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, 2002, pp. 1–10.

[65] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 14–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337226