Information Retrieval and Spectrum Based Bug Localization: Better Together

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

ABSTRACT

Debugging often takes much effort and resources. To help developers debug, numerous information retrieval (IR)-based and spectrum-based bug localization techniques have been proposed. 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). Both eventually generate a ranked list of program elements that are likely to contain the bug. However, these techniques only consider one source of information, either bug reports or program spectra, which is not optimal. To deal with the limitation of existing techniques, in this work, we propose a new multi-modal technique that considers both bug reports and program spectra to localize bugs. Our approach *adaptively* creates a bug-specific model to map a particular bug to its possible location, and introduces a novel idea of *suspicious words* that are highly associated to a bug. We evaluate our approach on 157 real bugs from four software systems, and compare it with a state-of-the-art IR-based bug localization method, a state-of-the-art spectrum-based bug localization method, and three state-of-the-art multi-modal feature location methods that are adapted for bug localization. Experiments show that our approach can outperform the baselines by at least 47.62%, 31.48%, 27.78%, and 28.80% in terms of number of bugs successfully localized when a developer inspects 1, 5, and 10 program elements (i.e., Top 1, Top 5, and Top 10), and Mean Average Precision (MAP) respectively.

1. INTRODUCTION

Developers often receive a high number of bug reports [9] and debugging these reports is a difficult task that consumes much resources [45]. To help developers debug, many research studies have proposed a number of techniques that 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, which are 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 can be divided into two families: information retrieval (IR)-based bug localization techniques [39, 44, 57, 41], and spectrum-based bug localization techniques [19, 8, 40, 56, 55, 13, 23, 24, 25]. IR-based bug localization techniques typically analyze textual descriptions contained in bug reports and identifier names and comments in source code files. It then returns a ranked list of program elements (typically program files) that are the most similar to the bug textual description. Spectrum-based bug localization techniques typically analyze program spectra that corresponds to program elements that are executed by failing and successful execution traces. It then returns 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 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 plan to address the limitation of existing studies by analyzing both bug reports and execution traces. We refer to the problem as *multi-modal bug localization* since we need to consider multiple modes of inputs (i.e., bug reports and program spectra). It fits well to developer debugging activities as illustrated by the following debugging 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 cannot be replicated, he will mark the bug report as "WORKSFORME" and will not continue further [33]. He will only proceed to the debugging step after the bug has been successfully replicated. 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.

Although no multi-modal bug localization technique has been proposed in the literature, there are a few multi-modal feature location techniques. These techniques process both feature description and program spectra to recommend program elements (typically program methods) that implement a corresponding feature [37, 26, 16]. These feature location approaches can be adapted to locate buggy program elements by replacing feature descriptions with bug reports and feature spectra with buggy program spectra. Unfortunately, our experiment (see Section 5) shows that the performance of such adapted approaches are not optimal yet.

Our multi-modal bug localization approach improves previous multi-modal approaches based on two intuitions. First, we note that there are a wide variety of bugs [46, 49] and different bugs often require different treatments. Thus, there is a need for a bug localization technique that is adaptive to different types of bugs. Past approaches [26, 16, 37] propose a one-size-fits-all solution. Here, we propose an instance-specific solution that considers each bug individually and tunes various parameters based on the characteristic of the bug. Second, Parnin and Orso [35] highlight in their study that some words are useful in localizing bugs and suggest that "future research could also investigate ways to automatically suggest or highlight terms that might be related to a failure". Based on their observation, we design an approach that can automatically highlight *suspicious words* and use them to localize bugs.

Our proposed approach, named Adaptive Multi-modal bug Localization (AML), realizes the above mentioned intuitions. It consists of three components: AML^{Text}, $\text{AML}^{\text{Spectra}}$, and $\text{AML}^{\text{SuspWord}}$. AML^{Text} only considers the textual description in bug reports, and $\text{AML}^{\text{Spectra}}$ only considers program spectra. On the other hand, $\text{AML}^{\text{SuspWord}}$ takes into account suspicious words learned by analyzing textual description and program spectra together. $\text{AML}^{\text{SuspWord}}$ computes the *suspicious scores* of words that appear as comments or identifiers of various program elements. It associates a program element to a set of words and the suspiciousness of a word can then be computed based on the number of times the corresponding program elements appear in failing or correct execution traces. Each of these components would output a score for each program element, and AML computes the weighted sum of these scores. The final score is *adaptively* computed for each individual bug by tuning these weights.

We focus on localizing a bug to the method that contains it. Historically, most IR-based bug localization techniques find buggy files [39, 44, 57, 41], while most spectrum-based bug localization solutions find buggy lines [19, 8, 40]. Localizing a bug to the file that contains it is useful, however a file can be big and developers still need to go through a lot of code to find the few lines that contain the bug. Localizing a bug to the line that contains it is useful, however, a bug often spans across multiple lines. Furthermore, developers often do not have "perfect bug understanding" [35] 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. A method is not as big as a file, but it often contains sufficient context needed to help developers understand a bug.

We have evaluated our solution using a dataset of 157 real bugs from four medium to large software systems: AspectJ, Ant, Lucene, and Rhino. We collected real bug reports and real test cases from these systems. The test cases are run to generate program spectra. We have compared our approach against 3 state-of-the-art multi-modal feature localization techniques (i.e., PROMESIR [37], DIT^A and DIT^B [16]), a state-of-the-art IR-based bug localization technique [53], and a state-of-the-art spectrum-based bug localization technique [52]. We evaluated our approach based on two evaluation metrics: 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., [39, 44, 57, 41]. 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 [35]. MAP is a standard information retrieval metric to evaluate the effectiveness of a ranking technique [31]. Our experiment results highlight that, among the 157 bugs, AML can successfully localize 31, 71, and 92 bugs when developers only inspect the top 1, top 5, and top 10 methods in the lists that 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%.

We summarize our key contributions 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 in AML.

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 $\text{AML}^{\text{Spectra}}$) 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 have evaluated our approach 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 structure of the remainder of this paper is as follows. In Section 2, we describe preliminary materials about IR-based and spectrum-based bug localization. In Section 3, we present our proposed approach AML. In Section 5, we describe our experiment methodology and results. We discuss related work in Section 6. We conclude and describe future work in Section 7.

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. [41], 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 [36].

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:

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

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 ...

Figure 1: Bug Report 54460 of Apache Ant

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	$n_s(e)$	$n_s(\bar{e})$

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

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

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 [19]. 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. 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}, $\text{AML}^{\text{Spectra}}$, $\text{AML}^{\text{SuspWord}}$, and Integrator. AML^{Text} processes only the textual information in the input bug reports using an IR-based bug localization technique described in Section 2. AML^{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} outputs a score that indicates how close is m to b which is denoted as $\text{AML}^{\text{Text}}(b, m, C)$. By default, AML^{Text} uses VSM as the IR-based bug localization technique.

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

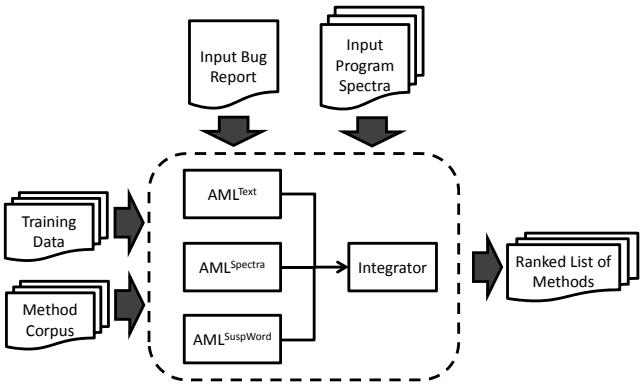

Figure 2: Proposed Approach: AML

$\text{AML}^{\text{SuspWord}}$ 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, $\text{AML}^{\text{SuspWord}}$ outputs a score that indicates how suspicious is m considering b and p; this is denoted as $\text{AML}^{\text{SuspWord}}(b, p, m, C)$.

The integrator component combines the AML^{Text}, $\text{AML}^{\text{Spectra}}$, $\text{AML}^{\text{SuspWord}}$ 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 $\text{AML}(b, p, m, C)$.

The AML^{Text} and $\text{AML}^{\text{Spectra}}$ components reuse techniques proposed in prior works which are described in Section 2. In the next subsections, we just describe the new components namely $\text{AML}^{\text{SuspWord}}$ 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" [35], 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 $\text{AML}^{\text{SuspWord}}$ 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.

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 com-

ments 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 $words(m)$.

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.

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:

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

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:

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

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 $\text{AML}^{\text{Spectra}}$ component. With this, $\text{AML}^{\text{SuspWord}}$ 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 Integrator Component

The integrator component serves to combine the scores produced by the three components AML^{Text}, $\text{AML}^{\text{Spectra}}$ and $\text{AML}^{\text{SuspWord}}$ 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:

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

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.

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., [34]:

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

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.

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:

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

Algorithm 1 Iterative parameter tuning

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 η, regularization parameter λ, maximum training iterations T_{max}

Ensure: Weight parameters α, β, γ

1: Initialize α, β, γ to zero
2: **repeat**
3: **for** each $n \in \{1, \ldots, N\}$ **do**
4: **if** $n \bmod 2 = 0$ **then** ▷ Draw a positive instance
5: Randomly pick i from $\{1, \ldots, N\}$ s.t. $y_i = 1$
6: **else** ▷ Draw a negative instance
7: Randomly pick i from $\{1, \ldots, N\}$ 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

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}.

4. EXTENSION FRAMEWORK

The original Adaptive Multi-modal bug Localization (AML) remains two main issues. Firstly, for each bug report b, the same parameter vector $\vec{\theta_b} = (\alpha, \beta, \gamma)$ is used for all methods m (see equation 3). In other words, all method m share the same parameter vector $\vec{\theta_b}$ of bug report when evaluating the relevancy of bug report b with method m and program spectra p. In particular, each method m not only considers the parameter of bug report but also share the parameter of method. Therefore, equation 3 does not accurately reflect the suspiciousness score of method m given bug report b and program spectra p.

4.1 Network Lasso

5. EXPERIMENTS

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

5.1 Dataset

We use a dataset of 157 bugs from 4 popular software projects to evaluate our approach against the baselines. 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 [14]. 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 [14]. 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 [33]. In this process, he is creating a failing test case. Unfortunately, not all test cases are documented and saved in the version control systems.

5.2 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 [57, 41].
- **Mean Average Precision (MAP)**: MAP is an IR metric to evaluate ranking approaches [31]. 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 [39, 44, 57, 41].

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. The learning rate η and regularization parameter λ of AML are chosen by performing another cross validation on the training data, while the maximum number of iterations T_{max} is fixed as 30. We use $K = 10$ as default value for the number of nearest neighbors. We conduct experiments on an Intel(R) Xeon E5-2667 2.9GHz server running Linux 2.6.

We compare our approach against 3 state-of-the-art multi-modal feature localization techniques (i.e., PROMESIR [37], DIT^A and DIT^B [16]), a state-of-the-art IR-based bug localization technique named LR [53], and a state-of-the-art spectrum-based bug localization technique named MULTRIC [52]. We use the same parameters and settings that are described in their papers with the following exceptions that we justify. For DIT$^{\mathrm{A}}$ and DIT$^{\mathrm{B}}$, 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]" [16]. 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.

5.3 Research Questions

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

PROMESIR [37], SITIR [26], and several algorithm variants proposed by Dit et al. [16] are state-of-the-art multi-modal feature location techniques. Among the variants proposed by Dit et al. [16], 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 [53] (LR^A and LR^B) and MULTRIC [52] which are recently proposed state-of-the-art IR-based and spectrum-based bug localization techniques respectively.

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

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

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

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

Research Question 2: Are all components of AML contributing toward its overall performance?

To answer this research question, we simply drop one component (i.e., AML^{Text}, $AML^{SuspWord}$, and $AML^{Spectra}$) from AML one-at-a-time and evaluate their performance. In the process, we create three variants of AML: AML^{-Text}, $AML^{-SuspWord}$, and $AML^{-Spectra}$. To create AML^{-Text}, $AML^{-SuspWord}$, and $AML^{-Spectra}$, we exclude AML^{Text}, $AML^{SuspWord}$, and $AML^{Spectra}$ components from Equation 3 of our proposed AML, respectively. We use the default value of $K = 10$, and apply Algorithm 1 to tune weights of these variants, and compare their performance with our proposed AML.

Research Question 3: How effective is our Integrator component ?

Rather than using the integrator component, it is possible to use a standard machine learning algorithm, e.g., learning-to-rank, to combine the scores produced by AML^{Text}, $AML^{SuspWord}$, and $AML^{Spectra}$. 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} [18], which was also used by LR [53]. We simply replace the Integrator component with SVM^{rank} and evaluate the effectiveness of the resulting solution.

Research Question 4: How efficient is AML?

If AML takes hours to produce a ranked list of methods for a given bug report, then it would be less useful. In this research question, we investigate the average running time needed for AML to output a ranked list of methods for a given bug report.

Research Question 5: What is the effect of varying the number of neighbors K on the performance of 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. In particular, we want to investigate if the performance of AML remains relatively stable for a wide range of K.

5.4 Results

5.4.1 RQ1: AML vs. Baselines

Table 4 shows the performance of AML, and all the baselines in terms of Top N. Out of 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. This means that AML can successfully localize 47.62%, 31.48%, and 27.78% more bugs than the best baseline (i.e., PROMESIR) by investigating the top 1, top 5, and top 10 methods respectively.

Table 5 shows the performance of AML and the baselines in terms of MAP. AML achieves MAP scores of 0.187, 0.234, 0.284, and 0.243 for AspectJ, Ant, Lucene, and Rhino datasets, respectively. Averaging across the four projects, AML achieves an overall MAP score of 0.237 which outperforms all the baselines. AML improves the average MAP scores of PROMESIR, DIT^A, DIT^B, LR^A, LR^B, and MULTRIC by 28.80%, 100.85%, 111.61%, 451.16%, 91.34%, and 109.73% respectively. Moreover, considering each individual project, in terms of MAP, AML is still the best performing multi-modal bug localization approach. AML outperforms the MAP score of the best performing baseline, by 54.55%, 13.59%, 39.22%, and 19.70% for AspectJ, Ant, Lucene, and Rhino datasets, respectively.

Moreover, we find that our novel component of AML, i.e., $AML^{SuspWord}$, can outperform all the baselines. $AML^{SuspWord}$ can achieve a Top 1, Top 5, Top 10, and MAP scores of 26, 66, 83, and 0.193. These results outperform the best performing baseline by 23.81%, 22.22%, 15.28%, and 4.89% respectively.

5.4.2 RQ2: Contributions of AML Components

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

5.4.3 RQ3: Integrator vs. SVM^{rank}

Table 7 shows the results of comparing our Integrator with SVM^{rank}. We can note that for most subject programs and

Table 6: Contributions of AML Components

Approach	Top 1	Top 5	Top 10	MAP
AML$^{-\text{Text}}$	28	68	87	0.212
AML$^{-\text{SuspWord}}$	28	62	83	0.201
AML$^{-\text{Spectra}}$	26	63	84	0.210
AML	31	71	92	0.237

metrics, Integrator outperforms SVM^{rank}. This shows the benefit of our Integrator component which builds a personalized model for each bug and considers the data imbalance phenomenon.

Table 7: Integrator vs. SVM^{rank}.

Metrics	Project	Integrator	SVM^{rank}
Top 1	AspectJ	7	4
	Ant	9	7
	Lucene	11	10
	Rhino	4	4
	Overall	31	25
Top 5	AspectJ	13	11
	Ant	22	24
	Lucene	22	23
	Rhino	14	13
	Overall	71	71
Top 10	AspectJ	13	14
	Ant	31	31
	Lucene	29	26
	Rhino	19	16
	Overall	92	87
MAP	AspectJ	0.187	0.131
	Ant	0.234	0.234
	Lucene	0.284	0.267
	Rhino	0.243	0.227
	Overall	0.237	0.215

5.4.4 RQ4: Running Time

Table 8 shows means and standard deviations of AML's running time for different projects. From the table, we note that AML has an average running time of 46.01 seconds. Among the four projects, AML can process Rhino bugs with the least average running time (i.e., 17.94 seconds), and AML needs the longest running time to process AspectJ bugs (i.e., 72.79 seconds). Compared to the other three projects, AspectJ is considerably larger. Therefore, it takes more time for AML to tune its component weights. Considering that a developer can spend hours and even days to fix a bug [20], AML running time of 20-80 seconds is reasonable.

5.4.5 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., $K = \infty$). The results with varying numbers of neighbors is shown in Table 9. 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.

Table 8: Running Time of AML (seconds)

Project	Mean	Standard Deviation
AspectJ	72.79	5.50
Ant	40.88	2.52
Lucene	43.39	3.40
Rhino	17.94	1.58
Overall	46.01	18.48

Table 9: Effect of Varying Number of Neighbors (K)

#Neighbors	Top 1	Top 5	Top 10	MAP
$K = 5$	29	68	84	0.223
$K = 10$	31	71	92	0.237
$K = 15$	30	70	91	0.237
$K = 20$	29	70	88	0.227
$K = 25$	29	67	87	0.224
$K = \infty$	28	69	86	0.222

5.5 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 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 [39, 44, 57, 41]. MAP is also well known in the information retrieval community [31]. 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, 17, 15, 43]. 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.

6. 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 [37, 26, 16].

Poshyvanyk et al. proposed an approach named PROMESIR that computes weighted sums of scores returned by an IR-based feature location solution (LSI [32]) and a spectrum-based solution (Tarantula [19]), and rank program elements based on their corresponding weighted sums [37]. Then, Liu et al. proposed an approach named SITIR which filters program elements returned by an IR-based feature location solution (LSI [32]) if they are not executed in a failing execution trace [26]. 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 [16]. 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 [39, 30, 22, 44, 57, 41, 47, 48, 53].

Lukins et al. used a topic modeling algorithm named Latent Dirichlet Allocation (LDA) for bug localization [30]. Then, Rao and Kak evaluated the utility of many standard IR techniques for bug localization including VSM and Smoothed Unigram Model (SUM) [39]. In the IR community, historically, VSM is proposed very early (four decades ago by Salton et al. [42]), 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 [44]. 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 [57]. 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 [41]. 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 [47]. 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 filtering, class name similarity, bug fix recency, and bug fix frequency [53].

All these approaches can be used as the AML^{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. [53].

Spectrum-Based Bug Localization: Various spectrum-based bug localization approaches have been proposed in the literature [19, 8, 27, 28, 24, 25, 11, 12, 56, 55, 13, 29]. 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 [19]. 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 [27, 28]. Recently, Xie et al. conducted a theoretical analysis and found that several families of suspiciousness score formulas outperform other families [50]. Next, Yoo proposed to use genetic programming to generate new suspiciousness score formulas that can perform better than many human designed formulas [54]. Subsequently, Xie et al. theoretically compared the performance of the formulas produced by genetic programming and identified the best performing ones [51]. 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 [52]. MULTRIC has been shown to outperform the best performing formulas studied by Xie et al. [50] and the best performing formula constructed by genetic programming [54, 51].

Many of the above mentioned approaches that compute suspiciousness scores of program elements can be used in the $\text{AML}^{\text{Spectra}}$ 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 [37]. 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 [52].

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 [21]. Also, Rahman et al. used static bug-finding to improve the performance of statistical defect prediction and vice versa [38].

7. CONCLUSION AND FUTURE WORK

In this paper, we put forward a novel multi-modal bug localization approach named Adaptive Multi-modal bug Localization (AML). Different from previous multi-modal approaches that are one-size-fits-all, our proposed approach 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. AML (in par-

ticular its $AML^{SuspWord}$ component) also leverages the concept of *suspicious words* (i.e., words that are associated to a bug) to better localize bugs. We have evaluated our proposed approach on 157 real bugs from 4 software systems. 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, 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%.

In the future, we plan to improve the effectiveness of our proposed approach in terms of Top N and MAP scores. To reduce the threats to external validity, we also plan to investigate more bug reports from additional software systems.

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

8. 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. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 2009.

[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*, pages 35–39, 2005.

[10] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 361–370, 2006.

[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*, pages 49–60, 2010.

[12] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, 2010.

[13] 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*, pages 342–351, 2005.

[14] 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*, pages 433–436, 2007.

[15] 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*, pages 31–41, 2010.

[16] B. Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

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

[18] T. Joachims. Svm^{rank}: Support vector machine for ranking. `www.cs.cornell.edu/people/tj/svm_light/svm_rank.html`. Accessed: 2015-07-15.

[19] 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*, pages 273–282, 2005.

[20] S. Kim and E. J. W. Jr. How long did it take to fix bugs? In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*, pages 173–174, 2006.

[21] E. Kocaguneli, T. Menzies, and J. W. Keung. On the value of ensemble effort estimation. *IEEE Trans. Software Eng.*, 38(6):1403–1416, 2012.

[22] 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*, pages 364–367, 2013.

[23] 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*, pages 141–154, 2003.

[24] 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*, pages 15–26, 2005.

[25] 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*, pages 286–295, 2005.

[26] 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*, pages 234–243, 2007.

[27] 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*, pages 1–10, 2010.

[28] 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*, 26(2):172–219, 2014.

[29] 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*, pages 127–138, 2014.

[30] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information & Software Technology*, 52(9):972–990, 2010.

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

[32] 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*, pages 125–137, 2003.

[33] Mozilla. Bug fields. `https://bugzilla.mozilla.org/page.cgi?id=fields.html`. Accessed: 2015-03-16.

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

[35] 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*, pages 199–209, 2011.

[36] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[37] 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.*, 33(6):420–432, 2007.

[38] 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*, pages 424–434. ACM, 2014.

[39] 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*, pages 43–52, 2011.

[40] 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*, pages 30–39, 2003.

[41] 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*, pages 345–355, 2013.

[42] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[43] S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Trans. Software Eng.*, 39(4):552–569, 2013.

[44] B. Sisman and A. C. Kak. 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*, pages 50–59, 2012.

[45] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

[46] 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*, pages 271–280, 2012.

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

[48] 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*, pages 171–180, 2014.

[49] 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*, pages 200–203, 2013.

[50] 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.*, 22(4):31, 2013.

[51] 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*, pages 224–238, 2013.

[52] 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*, pages 191–200, 2014.

[53] 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, pages 689–699, 2014.

[54] 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*, pages 244–258, 2012.

[55] 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*, pages 1–10, 2002.

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

[57] 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 *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 14–24, 2012.