Network-Clustered Multi-Modal Bug Localization

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 debug, numerous information retrieval (IR)-based and spectrum-based bug localization techniques have 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 carrying out a joint optimization of bug localization error and clustering of both bug reports and program elements (i.e., methods). 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. To estimate the latent parameters of both bug reports and methods, NetML features an adaptive learning procedure based on Newton method that updates the parameters on a per-feature basis. 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 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 methods 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 [52]. 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 [45], [50], [65], [48], and *spectrum*-based bug localization techniques [21], [8], [47], [64], [63], [14], [28], [29], [30]. 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 [39]. 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 [45], [50], [65], [48], while most spectrum-based techniques find

buggy lines [21], [8], [47]. 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" [41] 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 [53], [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 [41] 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 Adaptive Multi-Modal Bug Localization (AML) [26]— which we extend in this journal paper. 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 important 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 more accurately. Secondly, NetML augments the network Lasso regularization [19] in its parameter learning procedure, which enforces similar bug reports (and methods) to have similar latent parameters. It must be noted that, different from the conventional network Lasso that deals with only a single network (graph), we impose regularization over two networks, i.e., bug report similarity and method similarity graphs. This allows us to achieve simultaneous clustering of bug reports and methods, and exploit the similarity traits to achieve a more effective bug localization.

To evaluate the efficacy of the NetML approach, we conducted experiments 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 were collected from these systems. The test cases were run to generate program spectra. We compare NetML with our previous AML method. Additionally, we evaluate our approach against three state-of-the-art multi-modal feature localization techniques (i.e., PROMESIR [42], DIT^A and DIT^B [17]), 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). We note that top N and MAP are widely used in past bug localization studies, e.g., [45], [50], [65], [48]. Our experiment results demonstrate that, among the 157 bugs, NetML 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 NetML produces, respectively. These constitute 48.39%, 15.49%, 8.7%, and 13.92% improvements over AML (which is the second best method in our benchmark), in terms of Top 1, Top 5 , Top 10, and MAP results respectively.

We summarize the key contributions of this paper below:

1) We present a novel multi-modal bug localization method that adaptively learns two sets of latent parameters that characterize each bug report and method, respectively. We are also the first to incorporate the network Lasso regularization on both bug report and method similarity networks, which facilitates an effective joint optimization of bug localization quality and clustering of both bug reports and methods.
2) We develop an adaptive learning procedure based on Newton update to jointly update the latent parameters of bug reports and methods on a per-feature basis. The procedure is based on the formulation of strict convex loss function, which provides a theoretical guarantee that any minimum found will be globally optimal.
3) We have extensively evaluated NetML on a dataset of 157 real bugs from four software systems using real bug reports and test cases. Our statistical significance tests reveal that NetML improves upon state-of-the-art bug localization approaches by a substantial margin.

The remainder of this paper is organized 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 this paper 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.

2.1 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 IR 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. [48], we also keep the original identifier names.
2) Stopword removal removes punctuation marks, special symbols, number literals, and common stopwords [6]. It also removes programming keywords such as *if*, *for*, *while*, etc., which usually appear too frequently to be useful 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 [51].

Numerous IR techniques 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) [44] 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:

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

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

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

$$sim(q,d) = \frac{\sum\limits_{w \in (q \bigcap d)} weight(w,q) \times weight(w,d)}{\sqrt{\sum\limits_{w \in q} weight(w,q)^2} \times \sqrt{\sum\limits_{w \in d} weight(w,d)^2}} \quad (1)$$

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.

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

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

2.2 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].

Based on this spectra, SBBL typically computes some raw statistics for every program element. Tables 1 and 2 summarize some raw statistics that can be computed for a program element e, given a program spectra p. These

TABLE 2
Raw Statistic Description

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

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 $n_s(e,p)$ by one unit. Similarly, if an unsuccessful test case executes program element e, then we increase $n_f(e,p)$ 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.

Different SBBL techniques have used different formulas to calculate the suspiciousness scores. Among these techniques, Tarantula is a popular one [21]. Using the notation in Table 2, the following is the formula that Tarantula uses to compute the suspiciousness score of program element e, given program spectra p:

$$Tarantula(e,p) = \frac{\frac{n_f(e,p)}{n_f(p)}}{\frac{n_f(e,p)}{n_f(p)} + \frac{n_s(e,p)}{n_s(p)}} \quad (2)$$

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

3 Proposed Approach

An overview of our NetML framework is given in Figure 2 (enclosed in the dashed box). NetML takes as input a new bug report as well as a *historical* set of previously localized bug cases, each comprising a bug report and a set of (faulty) methods along with their program spectra. If a method contains a root cause of the bug, it is labeled as *faulty* or *relevant*, otherwise it is labeled as *non-faulty* or *irrelevant*. Given a new bug report and its corresponding program spectra, NetML looks at the historical set of previously localized bugs, their corresponding spectra, as well as the relevant method corpus, and then produces a list of methods ranked based on their likelihood to be faulty.

NetML has three main components, namely: *feature extraction*, *graph construction*, and *integrator*. The feature extraction component serves to extract multi-modal input features

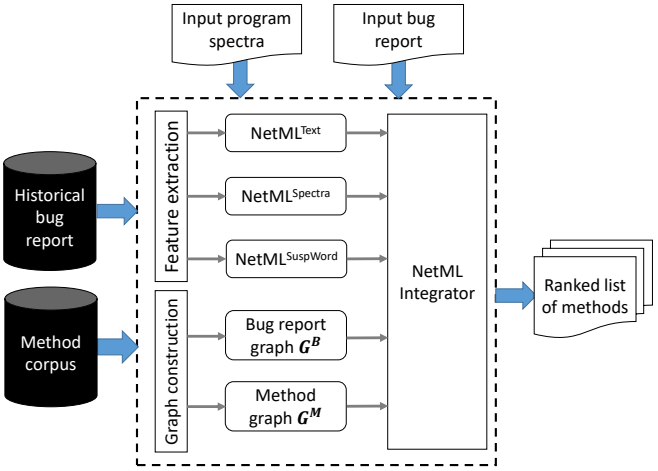

Fig. 2. The proposed NetML framework

that quantify different perspectives on the degree of relevancy between a bug report and a method. Note that this is in a similar spirit to [26]. Meanwhile, the graph construction component computes the similarity graphs among the bug reports ($\mathcal{G}_B$) and methods ($\mathcal{G}_M$).

Finally, the integrator component is the heart of NetML and constitutes the primary contribution of this work. It integrates both input features and similarity graph information in order to produce a ranked list of methods based on their relevancy score. In particular, the integrator performs adaptive learning that aims at jointly minimizing the bug localization errors and fostering clustering of the latent parameters of similar bug reports and/or methods.

In Sections 3.1–3.3, we first describe the NetML integrator component in greater details, including the formulation of our new integrator model as well as the corresponding objective function and adaptive learning procedure. We then elaborate the feature extraction and graph construction components in Sections 3.4 and 3.5, respectively.

3.1 Integrator Model

The new integrator model proposed in this work characterizes the relevancy of a method m to a given bug report b as an interaction between two types of latent parameters, namely: *latent bug report parameters* $\vec{u}_b = [u_{b,1}, \ldots, u_{b,j}, \ldots, u_{b,J}]$ and *latent method parameters* $\vec{v}_m = [v_{m,1}, \ldots, v_{m,j}, \ldots, v_{m,J}]$, where J is the total number of features. More specifically, the integrator model computes the relevancy score $\hat{f}_{b,m}$ as follows:

$$\hat{f}_{b,m} = \hat{f}(\vec{x}_{b,m}, \vec{u}_b, \vec{v}_m) = \sum_{j=1}^{J} (u_{b,j} + v_{m,j}) x_{b,m,j} \quad (3)$$

where $\vec{x}_{b,m} = [x_{b,m,1}, \ldots, x_{b,m,j}, \ldots, x_{b,m,J}]$ is the feature vector corresponding to a bug report–method pair (b,m).

It is worth mentioning that the above model constitutes a generalization of the AML integrator model that we previously developed [26]. In AML, the final relevancy score is computed based solely on the latent bug report parameters, and this set of parameters is shared by all methods for a

given bug report. On the other hand, the NetML integrator model accounts for not only the latent bug report parameters but also the latent method parameters. The addition of the latter parameters provides a greater degree of freedom/flexibility in quantifying the contribution of different methods to the localization of a given bug report.

3.2 Objective Function

Based on the above model formulation, we devise an objective function that guides the learning process of our integrator model. Specifically, we consider a joint optimization of bug localization quality and clustering of similar bug reports and methods, expressed by the loss function $\mathcal{L}$:

$$\mathcal{L} = \mathcal{L}_{\text{Entropy}} + \mathcal{L}_{\text{Ridge}} + \mathcal{L}_{\text{NetLasso}} \tag{4}$$

This consists of three components:

$$\begin{aligned}\mathcal{L}_{\text{Entropy}} =& -\sum_{b\in\mathcal{B}}\sum_{m\in\mathcal{M}} w_{b,m}\Big[y_{b,m}\ln(\sigma(\hat{f}_{b,m})) \\ &+ (1-y_{b,m})\ln(1-\sigma(\hat{f}_{b,m}))\Big] \end{aligned}\tag{5}$$

$$\mathcal{L}_{\text{Ridge}} = \frac{\alpha}{2}\sum_{j=1}^{J}\left[\sum_{b\in\mathcal{B}} u_{b,j}^2 + \sum_{m\in\mathcal{M}} v_{m,j}^2\right] \tag{6}$$

$$\begin{aligned}\mathcal{L}_{\text{NetLasso}} =& \frac{\beta}{2}\sum_{j=1}^{J}\Bigg[\sum_{(b,b')\in\mathcal{G}^B} e_{b,b'}(u_{b,j}-u_{b',j})^2 \\ &+ \sum_{(m,m')\in\mathcal{G}^M} e_{m,m'}(u_{m,j}-u_{m',j})^2\Bigg] \end{aligned}\tag{7}$$

where $\mathcal{B}$ and $\mathcal{M}$ are the sets of bug reports and methods respectively, $y_{b,m}$ is a binary label that indicates whether method m is relevant to bug report b ($y_{b,m}=1$) or not ($y_{b,m}=0$), and $\sigma(\hat{f}_{b,m}) = \frac{1}{1+\exp(-\hat{f}_{b,m})}$ is the logistic function [15]. Also, $w_{b,m}$ denotes the instance weight of a bug report–method pair (b,m), while $e_{b,b'}$ and $e_{m,m'}$ are the edge weights reflecting the degree of similarity between two bug reports b and b' and two methods m and m', respectively. Finally, $\alpha > 0$ and $\beta > 0$ are the user-defined parameters that control the strength of the ridge and network Lasso regularization, respectively.

Note that $\mathcal{L}_{\text{Entropy}}$ refers to the so-called cross-entropy loss [40], which provides an error measure of the bug localization process. Here $\mathcal{L}_{\text{Entropy}}$ can be interpreted as the discrepancy between the probability distribution of the predictive model $\hat{f}_{b,m}$ and that of the true label $y_{b,m}$ [40]. We also introduce the instance weight[1] $w_{b,m}$ in (5) to cater for the extremely *skewed* distribution of the relevant vs. irrelevant methods for a given bug report, which is a major challenge in bug localization process. That is, the number of relevant (faulty) methods is much smaller than that of irrelevant (non-faulty) ones. To address this, we configure $w_{b,m}$ in such a way that imposes a greater penalty for relevant instances being incorrectly predicted/classified than that for irrelevant ones. Specifically, we set $w_{b,m}$ as:

$$w_{b,m} = \begin{cases} \frac{1}{N_{\text{faulty}}}, & \text{if } y_{b,m}=1 \\ \frac{1}{N-N_{\text{non-faulty}}}, & \text{if } y_{b,m}=0 \end{cases} \tag{8}$$

1. An instance refers to a specific bug report–method pair (b,m)

where N is the total number of instances observed in the historical data, and N_{faulty} is the number of faulty instances.

Meanwhile, the ridge regularization $\mathcal{L}_{\text{Ridge}}$ serves to penalize large values of the latent parameters [40], which in turn helps mitigate the risk of data overfitting. From a probabilistic perspective, this corresponds to the Gaussian prior distribution for the latent parameters $u_{b,j}$ and $v_{m,j}$, with zero mean and inverse variance of α [26]. Finally, $\mathcal{L}_{\text{NetLasso}}$ refers to the network Lasso regularization [19], which enforces clustering of the latent parameters of bug reports and methods. The intuition is straightforward—the more similar two bug reports or two methods are (as quantified by $e_{b,b'}$ and $e_{m,m'}$), the closer their latent parameters $\vec{u}_b$ and $\vec{v}_m$ should be. This combination of $\mathcal{L}_{\text{Entropy}}$, $\mathcal{L}_{\text{Ridge}}$ and $\mathcal{L}_{\text{NetLasso}}$ facilitates a robust model that can simultaneously optimize the bug localization quality and cluster the latent parameters of similar bug reports and methods.

Next, in order to minimize the joint loss $\mathcal{L}$, we employ a Newton method [22] that is derived from a second-order Taylor series expansion of the loss function $\mathcal{L}$:

$$\mathcal{L}(\theta) = \mathcal{L}(\theta_0) + \nabla\mathcal{L}(\theta_0)(\theta-\theta_0) + \frac{\nabla^2\mathcal{L}(\theta_0)}{2}(\theta-\theta_0)^2 \tag{9}$$

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

$$\begin{aligned} 0 &= \nabla\mathcal{L}(\theta_0) + \nabla^2\mathcal{L}(\theta_0)(\theta-\theta_0) \\ \theta &= \theta_0 - \frac{\nabla\mathcal{L}(\theta_0)}{\nabla^2\mathcal{L}(\theta_0)} \end{aligned}\tag{10}$$

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

$$u_{b,j} \leftarrow u_{b,j} - \frac{\nabla\mathcal{L}(u_{b,j})}{\nabla^2\mathcal{L}(u_{b,j})} \tag{11}$$

$$v_{m,j} \leftarrow v_{m,j} - \frac{\nabla\mathcal{L}(v_{m,j})}{\nabla^2\mathcal{L}(v_{m,j})} \tag{12}$$

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

$$\begin{aligned}\nabla\mathcal{L}(u_{b,j}) =& \sum_{m\in\mathcal{M}}\Big[w_{b,m}(\sigma(\hat{f}_{b,m})-y_{b,m})x_{b,m,j}\Big] \\ &+ \alpha u_{b,j} + \beta\sum_{b'}\left[e_{b,b'}\left(u_{b,j}-u_{b',j}\right)\right] \end{aligned}\tag{13}$$

$$\begin{aligned}\nabla^2\mathcal{L}(u_{b,j}) =& \sum_{m\in\mathcal{M}}\Big[w_{b,m}\sigma(\hat{f}_{b,m})(1-\sigma(\hat{f}_{b,m}))x_{b,m,j}^2\Big] \\ &+ \alpha + \beta\sum_{b'} e_{b,b'} \end{aligned}\tag{14}$$

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

$$\begin{aligned}\nabla\mathcal{L}(v_{m,j}) =& \sum_{b\in\mathcal{B}}\Big[w_{b,m}(\sigma(\hat{f}_{b,m})-y_{b,m})x_{b,m,j}\Big] \\ &+ \alpha v_{m,j} + \beta\sum_{m'}\left[e_{m,m'}\left(v_{m,j}-v_{m',j}\right)\right] \end{aligned}\tag{15}$$

$$\begin{aligned}\nabla^2\mathcal{L}(v_{m,j}) =& \sum_{b\in\mathcal{B}}\Big[w_{b,m}\sigma(\hat{f}_{b,m})(1-\sigma(\hat{f}_{b,m}))x_{b,m,j}^2\Big] \\ &+ \alpha + \beta\sum_{m'} e_{m,m'} \end{aligned}\tag{16}$$

Algorithm 1 Adaptive learning of the NetML integrator

Inputs:

Set of K relevant historical bug reports $\mathcal{B}_K$ (i.e., $|\mathcal{B}_K| = K$)
Set of all methods $\mathcal{M}$, where $|\mathcal{M}| = M$
New bug report query b^* along with its features $\mathbf{X}_{b^*} = \{x_{b^*,m,j}\} \in \mathbb{R}^{1\times M\times J}$
Historical features $\mathbf{X} = \{x_{b,m,j}\} \in \mathbb{R}^{K\times M\times J}$
Historical labels $\mathbf{Y} = \{y_{b,m}\} \in \mathbb{R}^{K\times M}$
Bug report similarity graph $\mathcal{G}_B$, represented by the adjacency matrix $\mathbf{E}_B = \{e_{b,b'}\}$
Method similarity graph $\mathcal{G}_M$, represented by the adjacency matrix $\mathbf{E}_M = \{e_{m,m'}\}$

Outputs:

Relevancy scores $\hat{f}_{b^*,m} \in \mathbb{R}^{1\times M}$ of the new bug report b^* to all methods m
Latent bug report parameters $\mathbf{U} = \{u_{b,j}\} \in \mathbb{R}^{(K+1)\times J}$
Latent method parameters $\mathbf{V} = \{v_{m,j}\} \in \mathbb{R}^{M\times J}$

1: Compute the union set of bug reports $\mathcal{B} \leftarrow \mathcal{B}_K \cup \{b^*\}$
2: Initialize all latent parameters $u_{b,j} \leftarrow 0$ and $v_{m,j} \leftarrow 0$, $\forall b \in \mathcal{B}, m \in \mathcal{M}, j \in \{1,\ldots,J\}$
3: Precompute all constant terms $q_b \leftarrow \sum_{b'} e_{b,b'}$ and $q_m \leftarrow \sum_{m'} e_{m,m'}$, $\forall b \in \mathcal{B}, m \in \mathcal{M}$
4: Compute the bug probabilities $\sigma(\hat{f}_{b,m})$ for all (b, m) pairs via equation (3)
5: $\mathcal{L}_{\text{curr}} \leftarrow -\sum_b \sum_m w_{b,m}\big[y_{b,m} \ln\big(\sigma(\hat{f}_{b,m})\big) + \big(1 - y_{b,m}\big) \ln\big(1 - \sigma(\hat{f}_{b,m})\big)\big]$
6: **repeat**
7: $\quad\mathcal{L}_{\text{prev}} \leftarrow \mathcal{L}_{\text{curr}}$
8: $\quad$**for** each $j \in \{1,\ldots,J\}$ **do**
9: $\quad\quad$**/* Update the latent bug report parameters $u_{b,j}$ */**
10: $\quad\quad$**for** each $b \in \mathcal{B}$ **do**
11: $\quad\quad\quad p_b \leftarrow \sum_{b'} e_{b,b'} u_{b',j}$
12: $\quad\quad$**end for**
13: $\quad\quad$**for** each $b \in \mathcal{B}$ **do**
14: $\quad\quad\quad u_{\text{numer}} \leftarrow \sum_m \big[w_{b,m}(\sigma(\hat{f}_{b,m}) - y_{b,m})x_{b,m,j}\big] + \beta\big[u_{b,j}q_b - p_b\big] + \alpha u_{b,j}$
15: $\quad\quad\quad u_{\text{denom}} \leftarrow \sum_m \big[w_{b,m}\sigma(\hat{f}_{b,m})(1 - \sigma(\hat{f}_{b,m}))x_{b,m,j}^2\big] + \beta q_b + \alpha$
16: $\quad\quad\quad u_{b,j} \leftarrow u_{b,j} - \eta\left(\frac{u_{\text{numer}}}{u_{\text{denom}}}\right)$
17: $\quad\quad$**end for**
18: $\quad\quad$**/* Update the latent method parameters $v_{m,j}$ */**
19: $\quad\quad$**for** each $m \in \mathcal{M}$ **do**
20: $\quad\quad\quad p_m \leftarrow \sum_{m'} e_{m,m'} v_{m',j}$
21: $\quad\quad$**end for**
22: $\quad\quad$**for** each $m \in \mathcal{M}$ **do**
23: $\quad\quad\quad v_{\text{numer}} \leftarrow \sum_b \big[w_{b,m}(\sigma(\hat{f}_{b,m}) - y_{b,m})x_{b,m,j}\big] + \beta\big[v_{m,j}q_m - p_m\big] + \alpha v_{m,j}$
24: $\quad\quad\quad v_{\text{denom}} \leftarrow \sum_b \big[w_{b,m}\sigma(\hat{f}_{b,m})(1 - \sigma(\hat{f}_{b,m}))x_{b,m,j}^2\big] + \beta q_m + \alpha$
25: $\quad\quad\quad v_{m,j} \leftarrow v_{m,j} - \eta\left(\frac{v_{\text{numer}}}{v_{\text{denom}}}\right)$
26: $\quad\quad$**end for**
27: $\quad$**end for**
28: $\quad$Compute the updated bug probabilities $\sigma(\hat{f}_{b,m})$ via equation (3)
29: $\quad\mathcal{L}_{\text{curr}} \leftarrow -\sum_b \sum_m w_{b,m}\big[y_{b,m} \ln\big(\sigma(\hat{f}_{b,m})\big) + \big(1 - y_{b,m}\big) \ln\big(1 - \sigma(\hat{f}_{b,m})\big)\big]$
30: $\quad\eta \leftarrow \begin{cases} \frac{\eta}{2}, & \text{if } \mathcal{L}_{\text{curr}} > \mathcal{L}_{\text{prev}} \\ \min(1, 2\eta), & \text{otherwise} \end{cases}$
31: **until** T_{max} iterations
32: Compute the relevancy scores $\hat{f}_{b^*,m}$ using equation (3)

Finally, the update formula for $u_{b,j}$ can be obtained by substituting equation (13) and (14) into equation (11). Likewise, we can substitute (15) and (16) into (12) to arrive at the update formula for $v_{m,j}$. To learn the latent parameters, we use a *Newton method* that updates the parameters on a per-feature j basis. This will be elaborated in Section 3.3.

3.3 Adaptive Learning

Algorithm 1 summarizes the adaptive learning procedure of the NetML integrator for computing the relevancy scores of a new bug report (i.e., a new query) to different methods (i.e., documents). Given a new bug report b^*, the set of K relevant bug reports $\mathcal{B}_K$ in the historical data, the set of all methods $\mathcal{M}$, and the similarity graphs $\mathcal{G}_B$ and $\mathcal{G}_M$, the learning procedure appends b^* into $\mathcal{B}_K$ and then updates the latent parameters on a per-feature basis. That is, for each feature j, it performs Newton updates on the latent bug report parameters $u_{b,j}$ (steps 14–16) and latent method parameters $v_{m,j}$ (steps 23–25), in accordance with equations (11) and (12) respectively. The procedure is repeated until a

maximum iteration T_{max} is reached. Afterwards, the final prediction score $\hat{f}_{b^*,m}$ of the new bug report b^* for each method m is computed via equation (3).

Note that the selection of relevant bug report set B_K is based on the K-nearest neighbor retrieval from the bug report similarity graph $\mathcal{G}_B$, as follows:

$$\mathcal{B}_K = \text{Top}_K(\{e_{b^*,b} \mid \forall b \neq b^*\}) \tag{17}$$

where Top_K is a function that returns bug reports with the highest similarity $e_{b^*,b}$ to the query bug report b^*. The calculation of the similarity graphs is based on the VSM model and will be further described in Section 3.5.

It is also worth mentioning that the magnitude of the Newton update is downscaled by an adaptive learning rate η (where $0 < \eta \leq 1$). We introduce this scaling factor as a way to address the problem of *overshooting* in Newton method [12], whereby the update $\frac{\nabla\mathcal{L}(u_{b,j})}{\nabla^2\mathcal{L}(u_{b,j})}$ or $\frac{\nabla\mathcal{L}(v_{m,j})}{\nabla^2\mathcal{L}(v_{m,j})}$ is overestimated—possibly by many orders of magnitude. This may lead to oscillations and sometimes divergence in the loss function. To alleviate this issue, we compare the loss function $\mathcal{L}$ before and after a Newton iteration (step 30), and then adjust η accordingly depending on whether $\mathcal{L}$ increases or not. If it increases, then we reduce η by half in order to dampen the update magnitude; otherwise, the value of η gets doubled, up to a maximum limit of 1.

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. Additionally, during each Newton iteration, we have separate loops to compute the terms $\sum_{b'} e_{b,b'}u_{b',j}$ (step 11) and $\sum_{m'} e_{m,m'}v_{m',j}$ (step 20) for each feature j, prior to updating $u_{b,j}$ and $v_{m,j}$. The purpose is to make sure that, during the parameter updates (steps 14 and 23), the computation of $\sum_{b'} e_{b,b'}u_{b',j}$ in equation (11) and $\sum_{m'} e_{m,m'}v_{m',j}$ in equation (12) is based on the old parameter values from the previous iteration, and not affected by the ordering of b or m in the update loops.

We additionally highlight that the loss function $\mathcal{L}$ is *strictly convex*. This provides a nice theoretical guarantee that there is only one unique minimum in the loss function surface, and this minimum is globally optimal [46]. The convexity trait can be proven by looking at the curvatures (i.e., second derivatives) with respect to the latent bug report and method parameters, as per equations (14) and (16) respectively. Clearly, since $0 \leq \sigma(\hat{f}_{b,m}) \leq 1$, $w_{b,m}$, $x^2_{b,m,j}$, $e_{b,b'}$ and $e_{m,m'}$ are non-negative, while α and β are positive, the curvatures will be positive. The positive curvatures correspond to the so-called *positive definite* Hessian matrix—a well-known property of a strictly convex function [46].

3.4 Feature Extraction

The second component of the NetML framework is the feature extraction module, which generates features $\mathbf{X} = \{x_{b,m.j}\}$ to be fed as inputs to the NetML integrator (see Fig. 2). In line with our earlier AML work [26], for each bug report–method pair (b, m), we compute a feature vector $\vec{x}_{b,m}$ that consists of three elements (i.e., $J = 3$):

$$\vec{x}_{b,m} = \left[\text{NetML}^{\text{Text}}_{b,m}, \text{NetML}^{\text{Spectra}}_{b,m}, \text{NetML}^{\text{SuspWord}}_{b,m}\right] \tag{18}$$

The three features are elaborated in turn below.

$\text{NetML}^{\text{Text}}_{b,m}$ makes use of the TF-IDF method [44] to estimate the similarity between methods and bug reports. In particular, given a method m and a bug report b, $\text{NetML}^{\text{Text}}_{b,m}$ computes the cosine similarity between the TF-IDF representation of the bug report text and that of the method codes, which is akin to the IR-based bug localization method (cf. Section 2.1). That is, $\text{NetML}^{\text{Text}}_{b,m}$ is given by:

$$\text{NetML}^{\text{Text}}_{b,m} = sim(b, m) \tag{19}$$

where $sim(b, m)$ is the cosine similarity as defined in (1).

$\text{NetML}^{\text{Spectra}}_{b,m}$ processes only the program spectra information using a spectrum-based bug localization technique described in Section 2.2. Given a program spectra p corresponding to bug report b and a method m in a corpus C, $\text{NetML}^{\text{Spectra}}_{b,m}$ gives a score that quantifies how suspicious m is given p. By default, $\text{NetML}^{\text{Spectra}}_{b,m}$ uses the Tarantula method as described in Section 2.2 (cf. equation (2)):

$$\text{NetML}^{\text{Spectra}}_{b,m} = Tarantula(m, p, C) \tag{20}$$

Finally, $\text{NetML}^{\text{SuspWord}}_{b,m}$ processes both bug reports and program spectra, and computes the suspiciousness scores of words to rank different methods. It breaks a method into its constituent words, computes the suspiciousness scores of these words, and then aggregates these scores back in order to arrive at the suspiciousness score of the method. Given a bug report b, a program spectra p, and a method m in a corpus C, $\text{NetML}^{\text{SuspWord}}_{b,m}$ measures how suspicious m is considering b and p, as follows:

$$\text{NetML}^{\text{SuspWord}}_{b,m} = \text{SS}_{\text{method}}(m, p) \times \frac{\sum_{w \in b \cap m} \text{SSTFIDF}(w, p, b, C) \times \text{SSTFIDF}(w, p, m, C)}{\sqrt{\sum_{w \in b} \text{SSTFIDF}(w, p, b, C)^2} \times \sqrt{\sum_{w \in m} \text{SSTFIDF}(w, p, m, C)^2}} \tag{21}$$

where $\text{SS}_{\text{method}}(m, p)$ is the suspiciousness scores of a method, which by default uses the same score as the $\text{NetML}^{spectra}(m)$ component, and $\text{SSTFIDF}(w, p, b, C)$ is the weight of a word w in document (i.e., bug report or method) d with corpus C given program spectra p:

$$\text{SSTFIDF}(w, p, d, C) = \text{SS}_{\text{word}}(w, p) \times \ln(f(w, d) + 1) \times \ln \frac{|C|}{|d_i \in C : w \in d_i|} \tag{22}$$

where $\text{SS}_{\text{word}}(w, p)$ is the suspiciousness score of a word w:

$$\text{SS}_{\text{word}}(w, p) = \frac{\frac{|EF(w,p)|}{|p.FAIL|}}{\frac{|EF(w,p)|}{|p.FAIL|} + \frac{|ES(w,p)|}{|p.SUCCESS|}} \tag{23}$$

In the above equation, $EF(w, p)$ is the set of execution traces in $p.F$ that contain a method in which the word w appears, while $ES(w, p)$ is the set of execution traces in $p.S$ that contain a method in which the word w appears. Further details of all these components can be found in [26].

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

3.5 Graph Construction

The final component of the NetML framework is the graph construction module, which serves to compute the similarity graphs among bug reports and methods, to be used in the K-nearest neighbor retrieval as well as the network Lasso regularization. In this work, we define the bug report similarity graph $\mathcal{G}_B$ as comprising edge weights that reflect the textual similarity between two bug reports. For a pair of bug reports b and b', we define the edge weight $e_{b,b'}$ as:

$$e_{b,b'} = sim(b, b') \tag{24}$$

where $sim(b, b')$ is the cosine similarity between the TF-IDF weights of the textual descriptions of b and b', as per (1).

Similarly, the method similarity graph $\mathcal{G}_M$ comprises a set of edge weights $e_{m,m'}$ that reflect the textual similarity between two methods m and m'. This is given by:

$$e_{m,m'} = sim(m, m') \tag{25}$$

where $sim(b, b')$ is the cosine similarity between the TF-IDF representations of the source codes of m and m'.

4 Experiments

In this section, we first describe the datasets and evaluation settings used in our experiments. We then present a list of research questions we want to address, and accordingly elaborate our experiment results. Finally, we discuss some potential threats to the validity of our approach.

4.1 Dataset

To evaluate our approach, we use a dataset of 157 bugs from four popular software projects. The four 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 was 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 were 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 [39]. 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 Metrics and Settings

To assess the effectiveness of a bug localization method, we employ two key metrics, namely: Top N and mean average precision (MAP). They are respectively described below:

- **Top N**: Given a bug report, if one of its corresponding faulty methods is in the top-N results, we consider that the bug is successfully localized. The Top N score of a bug localization method is the number of bugs it can successfully localize [65], [48].
- **Mean Average Precision (MAP)**: MAP is an IR metric to evaluate ranking approaches [36], and is computed by taking the mean of the *average precision* scores across all bug reports. The average precision of a single bug report is computed as:

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

  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. Here $P(k)$ is the precision at a given top k methods, which is computed as follows:

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

  Note that the MAP scores of existing bug localization methods are typically low [45], [50], [65], [48].

Our evaluation procedure is based on 10-fold *cross validation* (CV). That is, for each project, we divide the bug reports into ten (mutually exclusive) sets. Then, for each fold, we take 1 set as new bug report queries (i.e., testing set) and treat the remaining 9 sets as historical bug reports (i.e., training set). We repeat this 10 times, and then collate the results to get the aggregated Top N and MAP scores.

In all our experiments, the hyper-parameters of the NetML method were configured as follows. Firstly, the regularization parameters α and β were chosen by performing 10 fold cross validation on the training set. Next, the maximum number of iterations T_{max} was fixed to 30. We use $K = 10$ as default value for the number of nearest neighbors. Note that the NetML parameters K and T_{max} follow the settings used in AML [26], so as to ensure fair comparisons. All experiments were conducted on an Intel(R) Xeon 2.9GHz server running a Linux operating system.

In order to assess whether NetML substantially outperforms other bug localization methods, we apply *Wilcoxon signed-rank test* [56]; it is a non-parametric statistical significance test for comparing two related or matched samples, whereby the population cannot be assumed to be normally distributed. The Wilcoxon test was applied to two types of metric (i.e., TopN and MAP). For every evaluation metric,

we collated the 10-fold results of a bug localization technique across the four software projects (i.e., AspectJ, Ant, Lucene, and Rhino) and then performed the Wilcoxon test to compare the collated results of different techniques. For this test, our null hypothesis is that NetML performs *worse* than or *equal* to the other method, and so we used one-sided/tail p-value to validate this hypothesis. Moreover, we also apply the Benjamini-Hochberg (BH) [13] procedure to control the effect of multiple comparisons. If the p-value is sufficiently small (say, below a significance level of 0.05), we can confidently reject the null hypothesis and conclude that NetML is significantly better than the other method.

4.3 Research Questions

Our empirical study seeks to answer several research questions (RQ), as described in the following subsections.

4.3.1 RQ1: How Effective is NetML Compared to Other State-of-the-Art Techniques?

We compare our NetML approach with its predecessor, i.e., AML [26], and several other state-of-the-art techniques. Previously, PROMESIR [42], SITIR [31], and several variants developed by Dit et al. [17] were state-of-the-art multi-modal feature location techniques. Among the variants proposed by Dit et al. [17], the best performing ones were $IR_{LSI}Dyn_{bin}WM_{HITS}(h, bin)^{bottom}$ and $IR_{LSI}Dyn_{bin}WM_{HITS}(h, freq)^{bottom}$. In this paper, we refer to these variants as DIT$^{\text{A}}$ and DIT$^{\text{B}}$ respectively. Dit et al. had shown that these two variants outperform SITIR, and so we exclude SITIR from our study. We also compare NetML with a state-of-the-art IR-based bug localization method named LR [61], and a state-of-the-art spectrum-based bug localization method named MULTRIC [60]. Note that, unlike PROMESIR, DIT^{A}, DIT^{B}, and MULTRIC which locate buggy methods, LR locates buggy files. Thus, we convert the list of files that LR produces into a list of methods by using two heuristics: (1) to return methods in a file in the same order that they appear in the file; and (2) to 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.

For all the above-mentioned techniques, we used the same parameters and settings as described in the respective papers, with the following exceptions that we justify. For DIT$^{\text{A}}$ and DIT$^{\text{B}}$, 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]" [17]. In this paper, since we used 10-fold CV, rather than using 66% of all bugs, we used all bugs in the training data (i.e., 90% of all bugs) to tune the threshold. For PROMESIR, we also used 10-fold CV and applied a brute force approach to tune PROMESIR's component weights using a step of 0.05.

4.3.2 RQ2: Do Feature Components of NetML Contribute toward Its Overall Performance?

To answer this question, we conducted an ablation test by dropping one feature component (i.e., NetML$^{\text{Text}}$, NetML$^{\text{SuspWord}}$, or NetML$^{\text{Spectra}}$) one-at-a-time and evaluating the performance. In the process, we created three variants of NetML: All$^{-\text{Text}}$, All$^{-\text{SuspWord}}$, and All$^{-\text{Spectra}}$. That is, we excluded Text, SuspWord, and Spectra from all feature components, respectively (see also Fig. 2). We used the default value of $K = 10$, and applied the NetML adaptive learning procedure (i.e., Algorithm 1) to tune the latent parameters of these variants. As our baseline, we performed the same ablation test to the feature components of the AML method (i.e., AML$^{\text{Text}}$, AML$^{\text{SuspWord}}$, or AML$^{\text{Spectra}}$).

4.3.3 RQ3: How Effective is the NetML Integrator?

Instead of using the NetML integrator component (see Section 3.1), one may consider using a standard machine learning algorithm, such as the learning-to-rank method, to combine the scores produced by the three feature components. Indeed, state-of-the-art IR-based and spectrum-based bug localization techniques such as LR and MULTRIC are based on the learning-to-rank method. As such, we conduct an experiment to compare our NetML integrator model with an off-the-shelf learning-to-rank model called SVM^{rank} [20], which was also used by LR [61]. To do so, we simply replace the NetML integrator model in Fig. 2 with SVM^{rank}, and then compare the resulting performances. For completeness, we also compare our NetML integrator with the integrator model used by the AML algorithm.

4.3.4 RQ4: What is the Effect of Varying the Number of Neighbors K on the Performance of NetML?

The most important parameter in our NetML approach is the number of nearest neighbors K (while the regularization parameters α and β were chosen via cross-validation—see Section 4.2). By default, we set the number of neighbors to $K = 10$, but the effect of varying this value is unclear. To answer this research question, we vary the value of K and investigate its effects on the performance of NetML. In particular, we wish to investigate if the performance remains relatively stable with varying values of K. For each K value, we also compare the performance of NetML against its predecessor (i.e., AML) using the same value.

4.4 Results

This section presents our experiment results and discussion in relation to the research questions raised in Section 4.3.

4.4.1 RQ1: Comparisons of NetML with Other Techniques

Table 4 shows the performance of NetML and all the other baseline methods including AML, in terms of the Top N score. Out of all 157 bugs, NetML can successfully localize 46, 82, and 100 bugs when the developers inspect the top 1, top 5, and top 10 methods respectively (see the "Overall" row). This implies that NetML can successfully localize 48.39%, 15.49%, and 8.7% more bugs than the best baseline (i.e., AML) by examining the top 1, top 5, and top 10 methods respectively. It is also shown that NetML outperforms the second best baseline (i.e., PROMESIR) by 119.05%, 51.85%, and 38.89% in terms of Top N.

Subsequently, Table 5 shows the MAP score of NetML along with those of the state-of-the-art multi-modal techniques. Averaging across the four projects, NetML achieves an overall MAP score of 0.270, which outperforms all the other baselines. In particular, NetML improves the average MAP performance of AML, PROMESIR, DIT$^{\text{A}}$, DIT$^{\text{B}}$, LR^{A},

TABLE 4
Top N results of different bug localization methods

Top N	Project	NetML	AML	PROMESIR	DIT^A	DIT^B	LR^A	LR^B	MULTRIC
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 (MAP) results of different bug localization methods

Project	NetML	AML	PROMESIR	DIT^A	DIT^B	LR^A	LR^B	MULTRIC
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

TABLE 6
The p-values of the Wilcoxon test applying the BH procedure on various pairs of bug localization methods

Method Comparison	Top 1	Top 5	Top 10	MAP
NetML vs. AML	0.029 (*)	0.033 (*)	0.079	0.006 (**)
NetML vs. PROMESIR	2×10^{-4} (**)	7×10^{-4} (**)	2×10^{-4} (**)	0.003 (**)
NetML vs. DIT^A	5×10^{-5} (**)	3×10^{-6} (**)	3×10^{-6} (**)	10^{-5} (**)
NetML vs. DIT^B	5×10^{-5} (**)	3×10^{-6} (**)	3×10^{-6} (**)	10^{-5} (**)
NetML vs. LR^A	8×10^{-4} (**)	5×10^{-6} (**)	3×10^{-6} (**)	5×10^{-5} (**)
NetML vs. LR^B	3×10^{-6} (**)	8×10^{-7} (**)	8×10^{-7} (**)	8×10^{-7} (**)
NetML vs. MULTRIC	9×10^{-6} (**)	5×10^{-6} (**)	8×10^{-7} (**)	5×10^{-6} (**)

(*): smaller than 0.05, (**): smaller than 0.01

LR^B, and MULTRIC by 13.92%, 46.74%, 128.81%, 141.07%, 527.91%, 112.60%, and 138.94% respectively. Considering the individual projects, in terms of MAP, NetML remains the best performing approach. In particular, NetML achieves MAP scores of 0.219, 0.270, 0.290, and 0.302 for the AspectJ, Ant, Lucene, and Rhino projects respectively. With respect to the best performing baseline (i.e., AML), these respectively constitute 17.11%, 15.38%, 2.11% and 24.28% improvements. Furthermore, NetML outperforms the MAP score of the second best baseline (i.e., PROMESIR) by 80.99%, 31.07%, 42.16%, and 48.77% across the four projects.

We finally performed the Wilcoxon test to compare the Top N and MAP results of different techniques. We again note that, for each evaluation metric, the Wilcoxon test was conducted on the results collated over the four software projects. Table 6 presents the p-values for different metrics (i.e., Top 1, Top 5, Top 10 and MAP), which were evaluated at the significance levels of 0.05 and 0.01. The results show that NetML significantly outperforms AML in three metrics (i.e., Top 1, Top 5, and MAP). Note that the success rates at Top 1 and Top 5 are more important than Top 10, since developers are less likely to check the 6th to 10th recommendations as compared to the first five recommendations [24]. Compared to the remaining techniques, NetML also performs significantly better in terms of Top 1, Top 5, Top 10 methods and MAP. Altogether, these results demonstrate the efficacy of our NetML approach.

4.4.2 RQ2: Contribution of Feature Components

Table 7 summarizes the results of our ablation test on both NetML and AML, each comparing the full model and three reduced variants (i.e., All$^{-\text{Text}}$, All$^{-\text{Text}}$ and All$^{-\text{Text}}$). It is evident that, for both NetML and AML, the full model always performs better than the reduced variants. This suggests that each feature component plays an important role, and omitting one of them may greatly affect the modelling performance. Among the three variants, it can be seen that All$^{-\text{SuspWord}}$ yields the smallest Top 1, Top 5, Top 10, and MAP scores for both NetML and AML. This suggests that the SuspWord feature component is more important than the other two (i.e., Text and Spectra).

Furthermore, comparing the results of NetML and AML, we can also observe that the former always gives a better, or at least equal, result than the latter. This suggests that the model parameterization using two sets of latent parameters

(instead of one in AML), along with the objective function formulation that jointly optimizes bug localization error and fosters clustering of similar bug reports and methods, contribute to a better overall performance of NetML.

4.4.3 RQ3: Comparisons among Integrator Models

Table 8 compares the performance of the NetML integrator model with that of the AML integrator and SVM^{rank}. We can observe that for all projects (i.e., AspectJ, Ant, Lucene, and Rhino) and metrics, the NetML integrator outperforms both the AML integrator and SVM^{rank}. With respect to SVM^{rank}, NetML achieves 84%, 15.49%, 14.94%, and 25.58% improvements, in terms of Top 1, Top 5, Top 10 and MAP scores across the four software projects, respectively. This can again be attributed to our NetML approach taking advantage of two sets of latent parameters and performing a joint optimization of bug localization error and clustering of similar bug reports and methods.

We also conducted the Wilcoxon test to examine whether the improvements over the AML integrator and SVM^{rank} are statistically significant. The resulting p-values are summarized in Table 9. As before, the NetML integrator significantly outperforms the AML integrator in terms of Top 1, Top 5 and MAP scores. It is also shown that the NetML integrator is significantly better than SVM^{rank} in different evaluation metrics (i.e., Top 1, Top 5, Top 10, and MAP). All in all, these justify the effectiveness of our NetML integrator component.

4.4.4 RQ4: Effect of Varying Number of Neighbors

To address this research question, we varied the number of nearest neighbors from $K = 5$ to all bug reports in the training set (i.e., $K = \infty$) for both NetML and AML. The results are shown in Table 10. We can see that, as we increase K, the performance of both multi-modal techniques improves until a certain point (i.e., $K = 15$), and decreases beyond that. This suggests that including more neighbors can improve performance to some extent. However, an overly large number of neighbors may lead to an increased level of noise (i.e., the number of irrelevant neighbors), resulting in a degraded performance. Nevertheless, the differences in the top N and MAP scores are fairly marginal, which justifies the robustness of our NetML approach. Looking at Table 10, it is also clear that NetML consistently outperforms AML for all K values (i.e., from $K = 5$ to $K = \infty$).

4.5 Discussion

There are several threats that may potentially impact the validity of our study. We discuss these threats below.

4.5.1 Number of Failed Test Cases and Its Impact

In our experiments with 157 bugs, most of the bugs were found to come with few failed test cases (average = 2.185). We have investigated if the number of failed test cases impacts the effectiveness of our approach. To this end, we computed 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 found that the differences are small (-0.472 to 0.074 test cases). These indicate that the number of test cases does not impact the effectiveness of our approach significantly and typically 1 to 3 failed test cases are sufficient for our approach to be effective.

4.5.2 Threats to Internal Validity

Threats to internal validity relate to implementation and dataset errors. We have checked our implementations and datasets. However, there could still 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.

5 Related Work

In this section, we highlight a number of research studies that are closely related to our work.

5.1 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], [31], [17].

Poshyvanyk et al. proposed an approach named PROMESIR that computes weighted sums of scores returned by an IR-based feature location solution (LSI [37]) and a spectrum-based solution (Tarantula [21]), 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 [37]) if they are not executed in a failing execution trace [31]. 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 [17]. 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.

5.2 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 [45], [35], [27], [50], [65], [48], [54], [55], [61].

Lukins et al. used a topic modeling algorithm named Latent Dirichlet Allocation (LDA) for bug localization [35]. Then, Rao and Kak evaluated the use of many standard IR methods for bug localization including VSM and Smoothed Unigram Model (SUM) [45]. In the IR community, VSM has a long history, proposed four decades ago by Salton et

TABLE 7
Contributions of feature components in NetML and AML

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

TABLE 8
Comparisons among different integrator models

Metrics	Project	NetML	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

TABLE 9
The p-values of the Wilcoxon test applying the BH procedure on various pairs integrator models

Method Comparison	Top 1	Top 5	Top 10	MAP
NetML vs. SVM^{rank}	0.003 (**)	0.041 (*)	0.033 (*)	0.008 (**)
NetML vs. AML	0.029 (*)	0.033 (*)	0.079	0.006 (**)

(*): smaller than 0.05, (**): smaller than 0.01

al. [49] and followed by many other IR methods including SUM and LDA, which address the limitations of VSM.

More recently, a number of approaches which consider information aside from text in bug reports to better locate bugs were proposed. Sisman and Kak proposed a version history-aware bug localization method that 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. [65] proposed an approach named BugLocator that includes a specialized VSM (named rVSM) and considers the similarities among bug reports to localize bugs. Next, Saha et al. [48] developed an approach that considers the structure of source code files and bug reports and employs structured retrieval for bug localization, and it outperforms BugLocator. 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 [54]. Most recently, Ye et al. devised 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 [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].

5.3 Spectrum-Based Bug Localization

Various spectrum-based bug localization approaches have been proposed [21], [8], [32], [33], [29], [30], [10], [11], [64], [63], [14], [34]. 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 [21]. 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 [32], [33]. 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

TABLE 10
Effect of varying the number of nearest neighbors on NetML and AML

#Neighbors	Top 1		Top 5		Top 10		MAP	
	NetML	AML	NetML	AML	NetML	AML	NetML	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

including the state-of-the-art spectrum-based approach by Xuan and Monperrus [60].

5.4 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 [38]. Also, Rahman et al. used static bug-finding in order to improve the performance of statistical defect prediction (and vice versa) [43]. Le et al. proposed SpecForge that combines different automaton based specification miners using model fission and model fusion in order to create a more effective specification miner [25]. Kellogg et al. presents *N-Prog* that combines static bug detection and test case generation to avoid unnecessary human effort [23]. In particular, N-Prog produces no false alarms, by construction, since its output alarm is either a new test case or a bug in a program.

6 Conclusion and Future Work

In this paper, we put forward a novel multi-modal bug localization approach named Network-clustered Multi-modal Bug Localization (NetML). Deviating from the contemporary multi-modal localization approaches, NetML is able to achieve an effective bug localization through the interplay of two sets of latent parameters characterizing both bug reports and methods. It also features an adaptive learning procedure that stems from a strictly convex objective function formulation, thereby provides a sound theoretical guarantee on the uniqueness of the optimal solution.

We have extensively evaluated NetML on 157 real bugs from four different software projects (i.e., AspectJ, Ant, Lucene, and Rhino). Among the 157 bugs, NetML is able to successfully localize 46, 82, and 100 bugs when developers inspect the top 1, top 5, and top 10 methods, respectively. Compared to the best performing baseline (i.e., AML), NetML can successfully localize 48.39%, 15.49%, and 8.7% more bugs when developers inspect the top 1, top 5, and top 10 methods, respectively. Furthermore, in terms of MAP, NetML outperforms the other baselines by 13.92%. Based on the Wilcoxon signed-rank test using BH procedure, we show that the results of NetML are significantly better across the four projects, in terms of top 1, top 10 and MAP scores.

Although NetML offers a powerful bug localization approach, there remains room for improvement. For example, the current approach as well as the IR-based techniques capture both bug report and program element (method) using a simple bag-of-words (e.g., TF-IDF) representation, ignoring the inherent structure within the source codes of a program, such as function call and/or data dependencies. In the future, we wish to consider a richer set of structural information within a program element, which carries additional semantics beyond the lexical terms. In particular, we would like to leverage both program structure information and lexical source code to localize potential bugs. We also plan to develop a more sophisticated technique, e.g., based on deep learning [18], to automatically learn the feature representation of bug reports and program elements.

Dataset and Codes. Additional information on the 157 bugs used in the experiments can be found at https://bitbucket.org/amlfse/amldata. The codes of the NetML method can also be made available upon request.

7 Acknowledgment

This research is supported by the National Research Foundation, Prime Ministers Office, Singapore under its International Research Centres in Singapore Funding Initiative.

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," *Journal of Systems and Software*, 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*, ser. Eclipse '05. New York, NY, USA: ACM, 2005, pp. 35–39. [Online]. Available: http://doi.acm.org/10.1145/1117696.1117704

[10] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831715

[11] 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 - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 265–274. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806840

[12] R. E. Bank and D. J. Rose, "Global approximate newton methods," *Numerische Mathematik*, vol. 37, no. 2, pp. 279–295, Jun. 1981. [Online]. Available: http://dx.doi.org/10.1007/BF01398257

[13] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995. [Online]. Available: http://dx.doi.org/10.2307/2346101

[14] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 342–351. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062522

[15] M. Collins, R. E. Schapire, and Y. Singer, "Logistic regression, adaboost and bregman distances," *Machine Learning*, 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 *Proceedings of the 20nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 433–436. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321702

[17] 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. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9194-4

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[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] T. Joachims, "Optimizing search engines using clickthrough data," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '02. New York, NY, USA: ACM, 2002, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/775047.775067

[21] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101949

[22] 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

[23] M. Kellogg, "Combining bug detection and test case generation," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ISSTA '16, Seattle, WA, USA,, 2016, pp. 1124–1126. [Online]. Available: http://doi.acm.org/10.1145/2950290.2983970

[24] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA '16. New York, NY, USA: ACM, 2016, pp. 165–176. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931051

[25] T. B. Le, X. D. Le, D. Lo, and I. Beschastnikh, "Synergizing specification miners through model fissions and fusions," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, Lincoln, NE, USA,, 2015, pp. 115–125. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.83

[26] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '15. New York, NY, USA: ACM, 2015, pp. 579–590. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786880

[27] T.-D. B. Le, S. Wang, and D. Lo, "Multi-abstraction concern localization," in *Proceeding of IEEE International Conference on Software Maintenance*, ser. ICSM '13, vol. 00. Los Alamitos, CA, USA: IEEE Computer Society, 2014, pp. 364–367.

[28] 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*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 141–154. [Online]. Available: http://doi.acm.org/10.1145/781131.781148

[29] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065014

[30] 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*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 286–295. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081753

[31] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the 20nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 234–243. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321667

[32] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2010.5609542

[33] 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. [Online]. Available: http://dx.doi.org/10.1002/smr.1616

[34] Lucia, D. Lo, and X. Xia, "Fusion fault localizers," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 127–138. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642983

[35] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2010.04.002

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

[37] 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*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776832

[38] T. Menzies, E. Kocaguneli, and J. W. Keung, "On the value of ensemble effort estimation," *IEEE Transactions on Software Engineering*, vol. 38, pp. 1403–1416, 2012.

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

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

[41] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 199–209. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001445

[42] D. Poshyvanyk, Y.-G. Guhneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval." *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007. [Online]. Available: http://dblp.uni-trier.de/db/journals/tse/tse33.html#PoshyvanykGMAR07

[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*, ser. ICSE '14. New York, NY, USA: ACM, 2014, pp. 424–434. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568269

[44] J. Ramos, "Using tf-idf to determine word relevance in document queries," Department of Computer Science, Rutgers University, 23515 BPO Way, Piscataway, NJ, 08855e, Tech. Rep., 2003.

[45] S. Rao and A. Kak, "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

[46] J. Renegar, *A Mathematical View of Interior-point Methods in Convex Optimization*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2001.

[47] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries." in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ser. ASE '03. IEEE Computer Society, 2003, pp. 30–39. [Online]. Available: http://dblp.uni-trier.de/db/conf/kbse/ase2003.html#RenierisR03

[48] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proceeding of 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '13, Silicon Valley, CA, USA, November 11-15, 2013, pp. 345–355. [Online]. Available: http://dx.doi.org/10.1109/ASE.2013.6693093

[49] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Magazine Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[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] K. Sparck Jones and P. Willett, Eds., *Readings in Information Retrieval*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

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

[53] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering*, ser. ISSRE '12, Dallas, TX, USA, November 27-30, 2012, pp. 271–280.

[54] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of 22nd International Conference on Program Comprehension*, ser. ICPC '14. New York, NY, USA: ACM, 2014, pp. 53–63. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597148

[55] S. Wang, D. Lo, and J. Lawall, "Compositional vector space models for improved bug localization," in *Proceedings of 30th IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14, Victoria, BC, Canada, 2014, pp. 171–180. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.39

[56] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[57] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An empirical study of bugs in software build systems," in *Proceedings of the 13th International Conference on Quality Software*, ser. ICQS '13, Najing, China, 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 Transactions on Software Engineering and Methodology*, vol. 22, no. 4, pp. 31:1–31:40, Oct. 2013. [Online]. Available: http://doi.acm.org/10.1145/2522920.2522924

[59] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation." in *Proceeding of the 5th International Symposium Search Based Software Engineering*, ser. SSBSE '13, G. Ruhe and Y. Zhang, Eds., vol. 8084. St. Petersburg, Russia: Springer, 2013, pp. 224–238. [Online]. Available: http://dblp.uni-trier.de/db/conf/ssbse/ssbse2013.html#XieKCYH13

[60] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14, Victoria, BC, Canada, 2014, pp. 191–200. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.41

[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*, ser. FSE '14, Hong Kong, China, 2014, pp. 689–699.

[62] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Proceedings of the 4th International Conference on Search Based Software Engineering*, ser. SSBSE '12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 244–258. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33119-0_18

[63] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10. New York, NY, USA: ACM, 2002, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/587051.587053

[64] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transaction on Software Engineering*, vol. 28, no. 2, pp. 183–200, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/32.988498

[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