Constrained Feature Selection for Localizing Faults

Tien-Duy B. Le[1], David Lo[1], and Ming Li[2,3]
[1]School of Information Systems, Singapore Management University, Singapore
[2]National Key Laboratory for Novel Software Technology, Nanjing University
[3]Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University
{btdle.2012,davidlo}@smu.edu.sg, lim@lamda.nju.edu.cn

***Abstract*—Developers often take much time and effort to find buggy program elements. To help developers debug, many past studies have proposed spectrum-based fault localization techniques. These techniques compare and contrast correct and faulty execution traces and highlight suspicious program elements. In this work, we propose *constrained* feature selection algorithms that we use to localize faults. Feature selection algorithms are commonly used to identify important features that are helpful for a classification task. By mapping an execution trace to a classification instance and a program element to a feature, we can transform fault localization to the feature selection problem. Unfortunately, existing feature selection algorithms do not perform too well, and we extend its performance by adding a constraint to the feature selection formulation based on a specific characteristic of the fault localization problem. We have performed experiments on a popular benchmark containing 154 faulty versions from 8 programs and demonstrate that several variants of our approach can outperform many fault localization techniques proposed in the literature. Using Wilcoxon rank-sum test and Cliff's d effect size, we also show that the improvements are both statistically significant and substantial.**

I. Introduction

Bugs always occur during software development, and debugging is a common task of developers to maintain software quality. However, it is also costly and time consuming for developers to manually find bugs in their programs. Recently, several fault localization techniques have been proposed to support developers in finding root causes of faults. One major family of fault localization techniques is referred to as spectrum-based fault localization (SBFL) [1], [2], [3]. These approaches take as input a set of passed and failed test cases, and analyze program spectra of successful and unsuccessful executions. Program spectra records which program elements are executed by a passing or failing test cases. Subsequently, SBFL techniques compute suspiciousness scores for program elements based on the program spectra. Various SBFL techniques employ different formulas to compute these scores. Suspiciousness scores reflect the likelihood of program elements to be buggy. Program elements are then sorted based on their suspiciousness scores. Using this ranked list, developers can manually inspect program elements from the beginning of the list until the bug has been found.

In this work, we propose an approach to localize faults by extending feature selection techniques. In machine learning, feature selection is a preprocessing step that retains relevant, important features of data instances that correlate to a particular class label, and excludes the irrelevant, redundant ones. In our approach, we model fault localization as a feature selection task, where program elements, test cases, and test outcomes correspond to features, data instances, and class labels, respectively. We use feature selection algorithms to identify important program elements (features) that correlate to a particular test outcome (class label). We are particularly interested in program elements that correlate to failures (failing test cases), since these elements are likely to be the locations where faults reside. We propose *constrained feature selection* (CFS) which extends standard feature selection to localize root causes of failures more accurately. CFS adjusts scores computed by a feature selection method by estimating how close a feature is to a class. Our approach uses these adjusted scores as suspiciousness scores of program elements. Similar to past studies, e.g., [4], we consider basic block as the granularity of program elements.

We have evaluated CFS to identify buggy basic blocks on a dataset of 154 faulty versions from 8 programs. These faults have been widely used to evaluate many SBFL studies [2], [1], [5], [4]. Using the Top-N score (i.e., number of faults successfully localized within top N suspicious program elements (i.e., basic blocks)) as the evaluation metric, our experiment results show that CFS statistically significantly and substantially outperforms popular (i.e., Tarantula [1] and Ochai [2]) and theoretically-best SBFL formulas [3]. We also compare CFS with standard feature selection methods, and find that CFS improves the effectiveness of standard feature selection methods for locating faults by 11.95% and 13.71% in terms of average Top-5 and Top-10 scores, respectively.

The contributions of our paper are listed as follows:

1) We propose a *constrained feature selection* approach which customizes standard feature selection techniques for localizing faults.
2) We evaluate our proposed approach on a dataset of 154 faults of 8 software programs and demonstrate that our approach statistically significantly and substantially outperforms many popular and theoretically-best SBFL formulas and standard feature selection techniques.

The rest of our paper is organized as follows: Section II describes background material on spectrum-based fault localization and feature selection. We present the details of the proposed approach and its evaluation in Section III and Section IV, respectively. Section V highlights related work. We conclude our paper and mention future work in Section VI.

II. Background

A. Spectrum-based Fault Localization Techniques

1) SBFL Concept: In a nutshell, spectrum-based fault localization helps developers find the location of a bug in a faulty program based on its passed and failed test cases. A SBFL technique returns a ranked list of program elements (e.g., statements, basic blocks, methods, files etc.) sorted by their suspiciousness scores. If a program element is more likely to be faulty, it is assigned with a higher suspiciousness score in the ranked list. Using the output list, developers can manually inspect program elements starting from the most suspicious element until the root cause of the fault is identified.

To determine suspiciousness scores of program elements, a SBFL technique first instruments a faulty program, and records execution traces by running an input set of passed and failed test cases. Each collected execution trace is a series of program elements invoked by a particular test case. Subsequently, a number of statistics are gathered from the execution traces for each program element. We refer to these statistics as SBFL statistics, which are shown in Table I. These statistics are eventually converted to suspiciousness scores by using a SBFL formula (presented in Section II-A2).

TABLE I
SBFL statistics of program element e

	Executed	**Not-Executed**
Failed	$n_f(e)$	$n_f(\bar{e})$
Passed	$n_p(e)$	$n_p(\bar{e})$

In Table I, $n_f(e)$ is the number of failed test cases that cover a program element e. $n_s(e)$ is the number of passed test cases that cover e. Similarly, $n_f(\bar{e})$ is the number of failed test cases that do not execute program element e, and $n_p(\bar{e})$ is the number of passed test cases that do not execute program element e. We also use other notations derived from the above four statistics: n_f is the total number of failed test cases (i.e., $n_f = n_f(e) + n_f(\bar{e})$), n_p is the total number of passed test cases (i.e., $n_p = n_p(e) + n_p(\bar{e})$).

2) SBFL Formulas: Each SBFL technique proposes its own formula to calculate suspiciousness scores from SBFL statistics. This section briefly describes two groups of SBFL formulas that are investigated in our study: well-known and theoretically-best formulas.

a) Well-Known Formulas: A number of SBFL formulas have been proposed to localize faults [2], [1], [5]. Among these formulas, Tarantula [1] and Ochiai [2] are the two popular ones. Using notations described in Table I and Section II-A1, Tarantula and Ochiai compute the suspiciousness score of a program element e as follows:

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

$$Ochiai(e) = \frac{n_f(e)}{\sqrt{n_f(n_f(e) + n_p(e))}}$$

According to the above formulas, both Tarantula and Ochiai assign non-zero suspiciousness scores to program elements that are executed by at least one failed test case (i.e., $n_f(e) > 0$).

b) Theoretically-Best Formulas: Recently, Xie et al. have discovered two families of SBFL formulas that are theoretically better than a number of other formulas, including well-known formulas such as Tarantula and Ochiai [3]. We refer to the two theoretically best SBFL families as ER1 and ER5. ER1 has two member formulas: ER1^a and ER1^b. ER5 has three members: ER5^a, ER5^b, and ER5^c. Using the notations shown in Table I and Section II-A1, the formulas belonging to the ER1 and ER5 families are:

$$ER1^a(e) = \begin{cases} -1 & \text{if } n_f(e) < n_f \\ n_p - n_p(e) & \text{if } n_f(e) = n_f \end{cases}$$

$$ER1^b(e) = n_f(e) - \frac{n_f(e)}{n_p(e) + n_p(\bar{e}) + 1}$$

$$ER5^a(e) = n_f(e)$$

$$ER5^b(e) = \frac{n_f(e)}{n_f(e) + n_f(\bar{e}) + n_p(e) + n_p(\bar{e})}$$

$$ER5^c(e) = \begin{cases} 0 & \text{if } n_f(e) < n_f \\ 1 & \text{if } n_f(e) = n_f \end{cases}$$

Under some assumptions, the above five formulas (i.e., ER1^a, ER1^b, ER5^a, ER5^b, and ER5^c) are theoretically proved to be the most effective ones among 30 SBFL formulas, including Ochiai and Tarantula [3]. However, in practice, using standard fault localization benchmark, Le et al. [6] demonstrate that the five theoretically best formulas can be outperformed by Ochiai if some assumptions do not hold.

B. Feature Selection Techniques

Feature selection is a preprocessing step for several machine learning tasks such as classification, clustering etc. It helps retain important, relevant features, and eliminate redundant, irrelevant ones. For classification, feature selection returns a subset of features that captures the most important and distinctive characteristics of instances in each class.

Feature selection algorithms are classified into three major families: filter, wrapper, and embedder. In this preliminary study, we investigate effectiveness of feature selection techniques that fall into the filter family for fault localization. We leave the investigation of wrapper and embedder algorithms for future work. Filter feature selection algorithms estimate a score for each feature based on a statistical measure (e.g., chi-squared statistics, information gain, etc.). Features with scores greater than a predefined threshold are included in the output of a filter feature selection algorithm. In our study, we consider the following algorithms [7]: chi-squared statistics, information gain, gain ratio, Relief, and Fisher score.

III. Proposed Approach

Overall Framework. In machine learning, feature selection outputs a subset of features that are relevant for a learning task.

Extending this concept, our study customizes feature selection for fault localization tasks, i.e., we output a ranked list of features (in our case: program elements) that captures the most important and distinctive characteristics of instances in a *specific class* (in our case: failed test cases). Our framework takes as input a faulty program and a set of passed and failed test cases. Using the given input, our approach collects execution traces from the faulty program by running the test cases one-by-one. Each execution trace consists of a series of executed program elements and we refer to it as a program spectrum *instance*. We extract a set of features from an instance, where each feature corresponds to a program element. The value of a feature is 1 if the corresponding program element is executed when the program is run with the corresponding test case, and 0 otherwise. An instance is labeled positive ("+1") if the corresponding test case is a failed test case, and negative ("−1") otherwise. A set of labeled instances along with their feature values are forwarded to the *constrained feature selection* module. This module then determines which features capture the most important and distinctive characteristics of failed test cases. Its output is a ranked list of program elements (i.e., features) sorted by their likelihood to be faulty. Subsequently, developers can manually inspect the ranked list from the most suspicious program element until the root cause of the fault is localized.

Constrained Feature Selection. In our study, we extend a standard feature selection method to identify program elements (i.e., features) that capture important characteristics of failed test cases. A standard feature selection method identifies features that can differentiate instances of different classes (in our case: passed test cases and failed test cases). Unfortunately, they do not differentiate features that characterize one class from those that characterize another. Two features can both have high discriminative power but characterize different classes. If a feature characterizes the positive class (i.e., failed test cases), a higher feature score indicates a higher likelihood for the corresponding program element to be faulty. However, if a feature characterizes the negative class (i.e., passed test cases), a higher feature score indicates a lower likelihood for the corresponding program element to be faulty. Thus, features deemed discriminative and important by a standard feature selection method may not necessarily be related to fault locations.

To overcome this issue, there is a need to separate important features based on how close they are to a class. We propose a constrained feature selection (CFS) method to solve this problem. CFS adjusts the scores computed by a feature selection method by estimating how close a feature is to a class. In order to determine the class that is closer to a feature, we use the following function:

$$RC(x) = \begin{cases} +1 & \text{if } \frac{n_f(x)}{n_f(x)+n_f(\bar{x})} > \frac{n_p(x)}{n_p(x)+n_p(\bar{x})} \\ -1 & \text{if } \frac{n_f(x)}{n_f(x)+n_f(\bar{x})} \leq \frac{n_p(x)}{n_p(x)+n_p(\bar{x})} \end{cases} \quad (1)$$

In the above function, the input parameter of RC is a feature x, which corresponds to a program element. $RC(x)$ is defined based on the notations of SBFL statistics (see Table I) to determine which class (i.e., positive or negative) that x is closer to. $\frac{n_f(x)}{n_f(x)+n_f(\bar{x})}$ is the ratio between the number of failed test cases that execute x and the total number of failed test cases. A higher value of this ratio indicates a stronger relationship between x and failed test cases. $\frac{n_p(x)}{n_p(x)+n_p(\bar{x})}$ is the ratio between the number of passed test cases that execute x and the total number of passed test cases. A higher value of this ratio indicates a stronger relationship between x and passed test cases. If $\frac{n_f(x)}{n_f(x)+n_f(\bar{x})}$ is strictly higher than $\frac{n_p(x)}{n_p(x)+n_p(\bar{x})}$, then x is closer to the positive class (i.e., failed test cases). Therefore, $RC(x)$ returns "+1". On the other hand, if $\frac{n_f(x)}{n_f(x)+n_f(\bar{x})}$ is less than or equal to $\frac{n_p(x)}{n_p(x)+n_p(\bar{x})}$, x is closer to the negative class (i.e., passed test cases). In this case, $RC(x)$ returns "−1".

Algorithm 1: Constrained Feature Selection for Localizing Faults

Input: $fprogram$: faulty program
TC: a set of failed and passed test cases
FS: a feature selection method
Output: Ranked list of program elements sorted by their likelihood to be faulty

```
1  Traces ← execute TC to collect execution traces
2  data ← extract features from Traces
   // run FS to estimate feature scores
3  scores ← FS(data)
4  suspiciousnessScores ← {}
5  for pe ∈ fprogram do
       // suspiciousness score of pe
6      susp ← scores[pe] × RC(pe)
7      suspiciousnessScores[pe] ← susp
8  end
   // construct the ranked list
9  rankedList ←
   sort(fprogram, suspiciousnessScores)
10 return rankedList
```

Algorithm 1 shows how CFS employs $RC(x)$ to adjust feature scores returned by a feature selection method. The input of the algorithm is a faulty program, a set of failed and passed test cases, and a feature selection method FS. The output of the algorithm is a ranked list of program elements sorted by their likelihood to be faulty. In the algorithm, lines 1 to 2 run the input test cases, and extract features from the collected execution traces. Line 3 runs a feature selection method FS to estimate feature scores of program elements. Lines 5 to 8 compute suspiciousness scores based on the feature scores and the return values of the $RC(x)$ function (see Equation 1). According to line 6, the suspiciousness score of a program element x is a product of the corresponding feature score and the return value of $RC(x)$. The $RC(x)$ function adjusts the signs of the scores of features (i.e., program elements) that are more relevant to the negative class (i.e., passed test cases). Therefore, features that are more important to the positive class (i.e., failed test cases) have higher suspiciousness

TABLE II
DATASET DESCRIPTION

Program	LOC	Language	# Faults	# Test Cases
print_token	478	C	5	4130
print_token2	399	C	10	4115
replace	512	C	31	5542
schedule	292	C	9	2650
schedule2	301	C	9	2710
tcas	141	C	36	1608
tot_info	440	C	19	1051
space	6,218	C	35	13,585

scores than those that are more important to the negative class. Subsequently, line 9 sorts program elements in descending order of suspiciousness scores. Program elements that appear at the beginning of the ranked list are the ones with highest feature scores among features that closer to the positive class (i.e., failed test cases). On the other hand, program elements that appear at the end of the ranked list are the ones with highest feature scores among features that are closer to the negative class (i.e., passed test cases). Finally, line 10 returns the ranked list of program elements.

IV. EXPERIMENTAL EVALUATION

A. Dataset

Table II describes a benchmark dataset that we use in this preliminary study to evaluate the effectiveness of fault localization techniques. It consists of multiple faulty versions of 8 C programs, i.e., Space, and 7 other programs obtained from the Siemens test suite, namely print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. Space is an interpreter for Array Definition Language (ADL) used in European Space Agency. Faults in the Siemens programs are seeded, while faults in Space are either real or seeded. We instrument basic blocks of these versions, and remove versions that our instrumentation technique is unable to reach (e.g., faults in global variable declarations). In total, we include 154 faulty versions of the 8 programs. These faulty versions have been widely used to evaluate the effectiveness of several spectrum-based fault localization techniques [2], [1], [5], [4].

B. Evaluation Metric & Experimental Settings

As evaluation metric, we use Top-N score, which is *number* of faults which can be successfully localized within top N most suspicious program elements. This score is motivated by the findings of Parnin and Orso [8]. They report that developers do not find fault localization to be useful if correct buggy elements do not appear among top-ranked program elements. Given a ranked list of program elements for localizing a fault, if one of the faulty program elements appears in the N most suspicious program elements, we classify the fault as successfully localized in the corresponding faulty program version. Note that ties are randomly broken. For example, if there are 20 program elements sharing the highest suspiciousness scores, we randomly select 10 of them to form the 10 most suspicious program elements. Since randomness is involved in the calculation of Top-N score, following Arcuri and Briand, we repeat the Top-N calculation process 1,000 times (we randomly break ties in different ways), and report the average Top-N score over the 1,000 iterations. In our experiments, we investigate $N \in \{1, 5, 10\}$.

TABLE III
AVERAGE TOP-N SCORE: CONSTRAINED AND STANDARD FEATURE SELECTION FOR FAULT LOCALIZATION. BEST SCORES ARE HIGHLIGHTED IN BOLD.

Approach	Average		
	Top-1	Top-5	Top-10
CFS$^{\texttt{CS}}$	20.903	68.305	90.908
CFS$^{\texttt{FS}}$	20.835	71.907	92.965
CFS$^{\texttt{GR}}$	21.390	69.673	90.569
CFS$^{\texttt{IG}}$	**22.746**	**72.475**	**95.000**
CFS$^{\texttt{RF}}$	5.201	27.308	46.125
CFS$^{\texttt{SU}}$	22.666	70.316	92.626
CS	20.727	61.752	80.331
FS	20.634	61.560	79.809
GR	21.295	62.978	81.316
IG	22.426	65.921	85.121
RF	3.748	23.564	37.530
SU	22.569	63.663	82.822

We investigate a number feature selection methods including Chi-Square (CS), Fisher Score (FS), Gain Ratio (GR), Information Gain (IG), Symmetrical Uncertainty (SU), and Relief (RF). The implementation of these methods are available in the the Weka toolkit[1] (v3.6.10) and Java-ML toolkit[2] (v0.1.7). We denote a variant of our proposed constrained feature selection technique built upon a standard feature selection method as CFS^{FS}, where FS is a symbol representing a feature selection method (i.e., CS, FS, GR, IG, SU, RF).

C. Results

1) RQ_1: How effective is our constrained feature selection technique for fault localization?

In this research question, we inspect six variants of our constrained feature selection technique for fault localization. Table III shows the average Top-N scores of the six CFS variants. We can note that CFS$^{\texttt{IG}}$, CFS$^{\texttt{SU}}$, and CFS$^{\texttt{FS}}$ achieve the best average top-N scores ($N \in \{1, 5, 10\}$).

2) RQ_2: Can constrained feature selection outperform existing SBFL techniques?

In this research question, we compare the effectiveness of CFS variants with well-known and theoretically best SBFL formulas (see Section II-A). Table IV shows the average Top-N scores of the seven SBFL formulas. We can note that Ochiai achieves the best average Top-1, 5, and 10 scores of 20.249, 69.852, and 91.601, respectively. Next, we perform the Wilcoxon rank-sum test [9] (at significance level of 1%) and compute Cliff's d effect size [10] to compare the average Top-N scores between the three most effective CFS variants (i.e., CFS$^{\texttt{IG}}$, CFS$^{\texttt{SU}}$, and CFS$^{\texttt{FS}}$) and Ochiai. We utilize R (3.2.1)[3] to perform the statistical tests.

Table V shows the result of Wilcoxon rank-sum tests and Cliff's d effect sizes. We find that the differences between average Top-N scores of all CFS variants and Ochiai are all statistically significant. Importantly, the Cliff's d effect sizes

[1] http://www.cs.waikato.ac.nz/ml/weka/index.html
[2] http://java-ml.sourceforge.net/
[3] http://www.r-project.org/

TABLE IV
AVERAGE TOP-N SCORES: WELL-KNOWN AND THEORETICALLY-BEST SBFL FORMULAS.

Approach	Average		
	Top-1	Top-5	Top-10
Ochiai	**20.249**	**69.852**	**91.601**
Tarantula	16.720	59.211	74.559
ER1^a	16.694	65.317	89.801
ER1^b	16.690	65.290	89.600
ER5^a	5.062	24.333	47.001
ER5^b	5.062	24.333	47.001
ER5^c	5.135	24.653	47.044

TABLE V
CLIFF'S D EFFECT SIZES AND WILCOXON RANK-SUM TEST RESULTS: CFS VARIANTS VS. OCHIAI. "*" INDICATES THAT THE DIFFERENCE BETWEEN THE AVERAGE TOP-N SCORES IS STATISTICALLY SIGNIFICANT (AT SIGNIFICANCE LEVEL OF 1%). "(TRIVIAL(T)/SMALL(S)/MEDIUM(M)/LARGE(L))" DENOTES THE CATEGORIZATION OF EFFECT SIZE.

CFS Variant	Ochiai		
	Top-1	Top-5	Top-10
CFS$^{\texttt{IG}}$	0.63* (L)	0.74* (L)	0.72* (L)
CFS$^{\texttt{FS}}$	0.18* (S)	0.36* (M)	0.65* (L)
CFS$^{\texttt{SU}}$	0.61* (L)	0.25* (S)	0.16* (S)

indicate that all of the differences between average Top-N scores are substantial (i.e., more than "trivial"). Overall, our proposed constrained feature selection approach statistically significantly and substantially outperforms popular and theoretically-best SBFL formulas.

3) RQ_3: Can constrained feature selection techniques outperform standard feature selection techniques for fault localization?

In this research question, we compare six variants of constrained feature selection (CFS) with standard feature selection methods: Chi-Square, Fisher Score, Gain Ratio, Information Gain, Relief, and Symmetrical Uncertainty. Table III shows the average Top-N scores of these methods. We can note that the mean of average Top-1 scores of standard feature selection methods are comparable to those of CFS variants. However, their Top-5 and Top-10 scores are lower; the mean of average Top-5 and Top-10 scores of the standard methods are approximately 11.95% and 13.71% lower than those of CFS. Wilcoxon rank-sum tests and Cliff's d effect sizes indicate that the differences of Top-5 and Top-10 scores are statistically significant and substantial (detailed results are not presented due to page limit). Overall, constrained feature selection is more effective in localizing faults than the standard methods.

D. Threats to Validity

We have checked our implementation and fixed any errors found, but there can still be errors that we are not aware of. We have analyzed 154 buggy versions from 8 programs written in the C language. These programs and buggy versions have been used to evaluate many past SBFL techniques [2], [1], [5], [4]. In the future, we plan to investigate more faults from more programs which are written in various programming languages. We employ the Top-N score, with different settings of N (i.e., $N \in \{1, 5, 10\}$), as the evaluation metric. Top-N score has been used as an evaluation metric in many other studies, e.g., [11], [12].

V. RELATED WORK

We have highlighted a number of SBFL techniques in Section II. Aside from these works, Roychowdhury and Khurshid propose an approach that also uses feature selection for fault localization [13]. Their approach employs Relief to localize faults. Our approach is different; instead of directly using scores computed by a feature selection method to localize faults, we adjust the scores by estimating how close a feature is to a class of test cases (i.e., passed or failed test cases). Our experimental results show that our proposed constrained feature selection (CFS) approach outperforms many standard feature selection methods in localizing faults, including Relief.

VI. CONCLUSION AND FUTURE WORK

In this work, we propose *constrained feature selection* (CFS) which customizes standard feature selection techniques for locating faults. We adjust scores computed by a feature selection technique by estimating how close a feature is to a class. Using the Top-N score as the evaluation metric, we have evaluated CFS on a dataset of 154 faulty versions from 8 programs. We find that CFS statistically significantly and substantially outperforms popular and theoretically-best SBFL formulas. Also, compared to standard feature selection methods, CFS improves their average Top-5 and Top-10 scores by 11.95% and 13.71% respectively.

As future work, we plan to investigate better ways to adapt existing feature selection techniques for fault localization, e.g., by refining Equation 1. We also plan to combine different CFS variants as well as current SBFL techniques to construct a more effective fault localization technique following composition techniques proposed in recent works [4], [14].

Acknowledgement. Ming Li is supported by NSFC (61422304,61272217) and JiangsuSF (BK20131278).

REFERENCES

[1] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.
[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*, 2007.
[3] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *TOSEM*.
[4] Lucia, D. Lo, and X. Xia, "Fusion fault localizers," in *ASE*, 2014.
[5] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.
[6] T.-D. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? a case with spectrum-based fault localization," in *ICSM*, 2013.
[7] I. Witten, E. Frank, and M. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kauffman Publisher, 2011.
[8] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *ISSTA*, 2011.
[9] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, pp. 80–83, 1945.
[10] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, 1993.
[11] 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 *International Conference on Software Engineering*, 2012.
[12] T. D. Le, R. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *ESEC/FSE*, 2015.
[13] S. Roychowdhury and S. Khurshid, "Software fault localization using feature selection," in *MALETS*, 2011.
[14] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *ICSME*, 2014, pp. 191–200.