Empir Software Eng (2015) 20:1237–1274
DOI 10.1007/s10664-014-9349-1

Should I follow this fault localization tool's output?

Automated prediction of fault localization effectiveness

Tien-Duy B. Le · David Lo · Ferdian Thung

Published online: 30 November 2014

Abstract Debugging is a crucial yet expensive activity to improve the reliability of software systems. To reduce debugging cost, various fault localization tools have been proposed. A spectrum-based fault localization tool often outputs an ordered list of program elements sorted based on their likelihood to be the root cause of a set of failures (i.e., their suspiciousness scores). Despite the many studies on fault localization, unfortunately, however, for many bugs, the root causes are often low in the ordered list. This potentially causes developers to distrust fault localization tools. Recently, Parnin and Orso highlight in their user study that many debuggers do not find fault localization useful if they do not find the root cause early in the list. To alleviate the above issue, we build an oracle that could predict whether the output of a fault localization tool can be trusted or not. If the output is not likely to be trusted, developers do not need to spend time going through the list of most suspicious program elements one by one. Rather, other conventional means of debugging could be performed. To construct the oracle, we extract the values of a number of features that are potentially related to the effectiveness of fault localization. Building upon advances in machine learning, we process these feature values to learn a discriminative model that is able to predict the effectiveness of a fault localization tool output. In this work, we consider an output of a fault localization tool to be effective if the root cause appears in the top 10 most suspicious program elements. We have evaluated our proposed oracle on 200 faulty versions of Space, NanoXML, XML-Security, and the 7 programs in Siemens test suite. Our experiments demonstrate that we could predict the effectiveness of 9 fault localization tools with a precision, recall, and F-measure (harmonic mean of precision and recall) of up

Communicated by: Yann-Gal Guhneuc and Tom Mens

T.-D. B. Le (✉) · D. Lo · F. Thung
School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore
e-mail: btdle.2012@smu.edu.sg

D. Lo
e-mail: davidlo@smu.edu.sg

F. Thung
e-mail: ferdiant.2013@smu.edu.sg

to 74.38 %, 90.00 % and 81.45 %, respectively. The numbers indicate that *many* ineffective fault localization instances are identified *correctly*, while only *few* effective ones are identified *wrongly*.

Keywords Fault localization · Effectiveness prediction · Classification

1 Introduction

Despite the advancement in software tools and processes, bugs are prevalent in many systems. In 2002, it was reported that software bugs cost US economy more than 50 billion dollars annually (Tassey 2002). Software testing and debugging cost itself is estimated to account for 30-90 % of the total labor spent on a project (Beizer 1990). Thus there is a need to develop automated means to help reduce software debugging cost. One important challenge in debugging is to localize the root cause of program failures. When a program fails, it is often hard to locate the faulty program elements that are responsible for the failure. The root cause could be located far from the location where the failure is exhibited, e.g., the location where a program crashes or produces a wrong output.

In order to address the high cost of debugging in general, and help in localizing root causes of failures in particular, many spectrum-based fault localization tools have been proposed in the literature, e.g., Jones and Harrold (2005), Abreu et al. (2007), and Lucia et al. (2010). These tools typically take in a set of normal execution traces and another set of faulty execution traces. Based on these set of program execution traces, these tools assign suspiciousness scores to various program elements. Next, program elements could be sorted based on their suspiciousness scores in descending order. The resultant list of suspicious program elements can then be presented to a human debugger to aid him/her in finding the root cause of a set of failures.

An *effective* fault localization tool would return a root cause at the top of a list of suspicious program elements. Although past studies have shown that fault localization tools could be effective for a number of cases, unfortunately, for many other cases, fault localization tools are not effective enough. Root causes are often listed low in the list of most suspicious program elements. Parnin and Orso pointed out in their user study that many developers do not find fault localization useful if they do not find the root cause early in the list (Parnin and Orso 2011). This *unreliability* of fault localization tools potentially cause many developers to distrust fault localization tools.

In this work, we plan to increase the usability of fault localization tools by building an oracle to predict if a particular output of a fault localization tool is likely to be effective or not. We define an output of a fault localization tool to be effective if the faulty program element or root cause is listed among the top-10 most suspicious program elements. With our tool, the debuggers could be better informed whether he can trust or distrust the output of a fault localization tool run on a set of program execution traces. The following scenarios illustrate the benefits of predicting the effectiveness of a fault localization output:

Scenario 1 - Without Oracle: Tien-Duy had 10 bugs to fix. He ran a fault localization tool for the 10 bugs. He followed the tool recommendations, however he only found 2 of the 10 recommendations to be effective. He wasted much time following 8 bad recommendations given by the tool.

Scenario 2 - With Oracle: Tien-Duy had 10 bugs to fix. He ran a fault localization tool for the 10 bugs and he had an oracle that can predict which fault localization outputs are likely to be effective. The oracle predicted that 3 outputs are likely to be effective. For 2 out of the 3 outputs, the fault localization outputs are indeed effective and saved Tien-Duy much time. Tien-Duy only wasted time following 1 bad recommendation.

To build the oracle, we extract values of important features from the execution traces and outputs of fault localization tools. These feature values extracted from a training data are then used to build a discriminative model leveraging a machine learning solution. The resultant discriminative model serves as an oracle and could be used to predict the effectiveness of a fault localization tool on other inputs. If fault localization instances are predicted to be effective, developers can continue to use these instances to localize bugs. Otherwise, developers could consider switching to other fault localization tools or traditional debugging. Thus, time and effort to follow a bad output of a fault localization tool can be saved.

We have experimented our approach on 200 faulty versions from NanoXML, XML-Security, Space, and the 7 programs in the Siemens test suite. We investigate a well known spectrum-based fault localization tool namely Tarantula (Jones and Harrold 2005) which was also studied by Parnin and Orso (2011). Our experiments show that we can predict whether Tarantula's outputs are effective or not by a precision, recall, and F-measure (i.e., harmonic mean of precision and recall) of 54.36 %, 95.29 %, and 69.23 %. We also investigate if our tool is effective to help eight other fault localization tools, i.e., Ochiai (Abreu et al. 2007), Information Gain (Lucia et al. 2010), $ER1^a$ (Xie et al. 2013), $ER1^b$ (Xie et al. 2013), $ER5^a$ (Xie et al. 2013), $ER5^b$ (Xie et al. 2013), $ER5^c$ (Xie et al. 2013), and DStar (Wong et al. 2014), with promising results. We could achieve a precision, recall, and F-measure of up to 74.38 %, 90.00 % and 81.45 %, respectively (for $ER1^a$ and $ER1^b$). Furthermore, we investigate whether a model trained based on one set of programs can be used on other programs. We perform cross-program validation in which we use data from a set of programs to learn a model and apply the model on a new program. Our experiments show that the F-measures of our approach in predicting the effectiveness of DStar, Ochiai, Information Gain, $ER1^a$, and $ER1^b$ in the cross-program setting are all above 70 %, which are reasonably high.

In our experiments, the time it takes to train a model is always less than one minute. Thus, when a new spectrum-based fault localization technique is analyzed, our approach only takes less than a minute to train a prediction model. The model can be used many times and thus effort invested to train a model is very minimal. Also, developers do not need to wait for a long time when using our approach as our trained models can compute the effectiveness of a fault localization instance in a fraction of a second.

In this work, our contributions are as follows:

1. We define a new research problem namely predicting the effectiveness of a fault localization tool given a set of execution traces. Solving this problem would help developers to better trust the output of a fault localization tool.
2. We present a machine learning framework to tackle the research problem. We propose a novel set of features that are relevant for predicting the effectiveness of a fault localization tool. We build upon and extend a state-of-the-art machine learning solution for the prediction problem by addressing the issue of imbalanced data. The issue of imbalanced data occurs since many outputs of Tarantula and several other fault localization tools are ineffective.
3. We have evaluated our approach on 200 faulty programs from NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite. We show that, for

Tarantula, we could achieve a precision, recall, and F-measure of 54.36 %, 95.29 %, and 69.23 %, respectively. This shows that *many* ineffective and *almost all* effective outputs of Tarantula are detected correctly. For the other 7 fault localization tools, we could achieve a precision, recall, and F-measure of up to 74.38 %, 90.00 % and 81.45 %, respectively (for $ER1^a$ and $ER1^b$).

The structure of this paper is as follows. In Section 2, we describe preliminary materials on spectrum-based fault localization and an intuition how effectiveness prediction could be solved. In Section 3, we present a birds-eye-view of our proposed framework. Section 4 outlines what features are extracted from the execution traces and output of the fault localization tool. Section 5 elaborates our approach to learn a discriminative model using a classification algorithm and how we address the problem of imbalanced data. We present our experiment settings, datasets, and results which answer a number of research questions in Section 6. We discuss related studies in Section 7. We finally conclude and mention future work in Section 8.

2 Preliminaries & Problem Definition

In this section, we first introduce fault localization. We then define the problem of effectiveness prediction and give some intuitions on how this could be solved.

2.1 Fault Localization

Fault localization takes as input a faulty program, along with a set of test cases, and a test oracle. The faulty program is instrumented such that when a test case is run over it, a program spectra is generated. A program spectra records certain characteristics of a particular program run and thus it becomes a behavioral signature of the run (Reps et al. 1997). This program spectra could constitute a set of counters which record how many times different program elements (e.g., statement, basic block, etc.) are executed in a particular program run (Harrold et al. 2000). Alternatively, the counter could record a boolean flag that indicates whether a program element is executed or not. The test oracle is used to decide if a particular program run is correct or faulty. Faulty runs or executions are also referred to as failures. Fault localization task is to analyze program spectra of correct and faulty runs with the goal of finding program elements that are the root causes of the failures (i.e., the faults or errors).

Various spectra have been proposed in past studies (Harrold et al. 2000). In this study, we use *block-hit spectra*; we instrument every block of a program and collect information on which blocks are executed in a run. Block-hit spectra is suitable as all statements in a basic block have the same execution profile. It has also been shown in the literature that the cost of collecting block-hit spectra is relatively low and the resultant spectra could be used for fault localization (Abreu et al. 2007; Harrold et al. 2000).

Figure 1 shows an example code with several program spectra. The identifiers of the basic blocks are shown in the first column. The statements located in the basic blocks are shown in the second column. There is a bug in the example code at basic block three; the condition of the if statement should be "count >= 1" instead of "count > 1". Columns 3 to 6 show the program spectra that are produced when four test cases are run. Three of the test cases do not expose the bug, i.e., running them result in correct executions. The fourth test case exposes the bug, i.e., running it result in a faulty execution. A cell marked

Blk ID	Program Elements	T1	T2	T3	T4
1	int count, n; Ele *proc; List *src_queue, *dest_queue; if (prio >= MAXPRIO) /*maxprio=3*/	●	●	●	●
2	{return;}	●			

3	src_queue = prio_queue[prio];
	dest_queue = prio_queue[prio+1];
	count = src_queue->mem_count;
	if (count > 1) /* Bug*//* supposed : count >=1*/ {
			●	●	●

4	n = (int) (count*ratio + 1);
	proc = find_nth(src_queue, n);
	if (proc) {
			●	●	

5	src_queue = del_ele(src_queue, proc);
	proc->priority = prio;
	dest_queue = append_ele(dest_queue, proc); } }
			●	●	

	Status of Test Case Execution :	P	P	P	F

Fig. 1 Four Block-Hit Program Spectra. T1, T2, T3, and T4 are the four test cases used in the example. The bullet indicates its corresponding program element is executed by a particular test case (i.e., T1, T2, T3, and T4). "P" and "F" stand for passed and failed execution of a test case, respectively

by a • indicates that a particular basic block is executed when a particular test case is run. An empty cell indicates that a particular basic block is not executed when a particular test case is run.

To identify the faulty program elements (e.g., basic block 3 in Fig. 1), we compute the suspiciousness scores of each of the program elements. There are various ways to define suspiciousness. In this work, we primarily consider a well-known suspiciousness score defined by Jones and Harrold, named Tarantula (Jones and Harrold 2005). Considering several notations in Table 1, Tarantula's suspiciousness score can be defined as follows:

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

Table 1 Spectra notations used in this work

Symbol	**Definition**
n	Total number of test cases in the test suite
n^e	Number of test cases that executes a program element e
n_s	Number of test cases that pass
n_f	Number of test cases that fail
n_s^e	Number of test cases that execute e and pass
n_f^e	Number of test cases that execute e and fail

Tarantula considers an element more suspicious if it occurs more frequently in failed executions than in correct executions. Considering the example shown in Fig. 1, the suspiciousness score of block 1 is: $\frac{1}{(1+1)} = 0.5$. The suspiciousness scores of block 2, 4, and 5 are zeros since the numerator of Tarantula (i.e., $\frac{n_f^e}{n_f}$) is zero. The suspiciousness score of block 3 is: $\frac{1}{(\frac{2}{3}+1)} = 0.6$. Thus using Tarantula, the most suspicious block is block 3, followed by block 1, followed by blocks 2, 4, and 5. We could sort the basic blocks based on their suspiciousness scores and the debugger could check the blocks one-by-one from the most to the least suspicious block. Following Tarantula's recommendation, the fault could be found after one basic block inspection.

Aside from Tarantula, there are several other fault localization studies that propose various suspiciousness score formulas including: Ochiai (Abreu et al. 2007), Information Gain (Lucia et al. 2010), $ER1^a$ (Xie et al. 2013), $ER1^b$ (Xie et al. 2013), $ER5^a$ (Xie et al. 2013), $ER5^b$ (Xie et al. 2013), $ER5^c$ (Xie et al. 2013), and DStar (Wong et al. 2014). We highlight them in the following paragraphs.

Ochiai. Abreu et al. propose Ochiai which has been shown to outperform Tarantula (Abreu et al. 2007). The formula for Ochiai is shown below:

$$Ochiai\ (e) = \frac{n_f^e}{\sqrt{n_f(n_f^e + n_s^e)}} = \frac{n_f^e}{\sqrt{n_f \times n^e}}$$

Information Gain. Lucia et al. perform a study of 20 association measures proposed in the data mining and machine learning community and highlight that information gain (Geng and Hamilton 2006) is the most effective measure for identifying suspicious program elements (Lucia et al. 2010). An association measure computes the strength of association between two variables: A and B. Information gain of A and B can be computed by the following formula:

$$\begin{aligned} IG(A, B) = (-P(B)\log P(B) - P(\overline{B})\log P(\overline{B})) - \\ (P(A) \times (-P(B|A)\log P(B|A)) - P(\overline{B}|A)\log P(\overline{B}|A) - \\ P(\overline{A}) \times (-P(B|\overline{A})\log P(B|\overline{A})) - P(\overline{B}|\overline{A})\log P(\overline{B}|\overline{A}))) \end{aligned}$$

In the above formula, $P(A)$ and $P(B)$ are the probabilities of A and B happening, respectively. $P(\overline{A})$ is the probability of A not happening. $P(A, B)$ is the probability of A and B happening together. $P(A|B)$ is the probability of A happening given that B happens. $P(B|A)$ is the conditional probability of B happening given that A happens.

Based on the above formula, let $IG(e, f)$ denote an information gain score between the executions of a program element e (e) and program failures (f). The suspiciousness score of a program element e is then defined as follows. If e is a control block (e.g., `while` or `if` statements), and $direct$ is the set of direct children of e in the control dependence graph of the containing program, the information-gain-based suspiciousness score of e (denoted as $InfoGain(e)$) is the maximum of the following:

1) $IG(e, f)$,
2) $max_{c \in direct} \cdot IG(c, f)$.

Otherwise, the suspiciousness score of e is $IG(e, f)$.

Theoretically best formulas. Xie et al. recently theoretically analyze 30 suspiciousness score formulas and prove that two families consisting of 5 formulas outperform the rest (Xie et al. 2013). The 5 formulas are named $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, and $ER5^c$. These formulas are shown below:

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

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

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

$$ER5^b(e) = \frac{n_f^e}{n_f^e + (n_f - n_f^e) + n_s^e + (n_s - n_s^e)}$$

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

Note that the above formulas have been shown to be theoretically the best *under certain conditions* (e.g., 100 % code coverage). The conditions are often not met in practice. Le et al. have shown that the performance of these formulas although are good, but are not the best on real datasets (Le et al. 2013).

DStar. Wong et al. propose a spectrum-based fault localization technique named *DStar* (Wong et al. 2014). They modify the formula of Ochiai (which is also known as *Kulczynski coefficient*) to obtain the following formula of DStar:

$$DStar(e) = \frac{(n_f^e)^*}{(n_f - n_f^e) + n_s^e}$$

In the above formula, * is an exponent (or power) of n_f^e whose value is greater than or equal to 1. Wong et al. show that DStar outperforms many spectrum-based fault localization techniques including Tarantula and Ochiai. They set * equals to 2, and find that the effectiveness of DStar increases until it levels off as the value of * is increased. In our study, we set * equals to 3.

2.2 Effectiveness Prediction

The goal of our work is to predict if a particular fault localization tool is effective for a particular set of execution traces. We refer to the process where a fault localization tool is used to process a set of execution traces and output a list of suspicious program elements as a *fault localization instance*. In our approach, we define a fault localization instance to be effective if the root cause is located among the top-10 most suspicious program elements. Ties are randomly broken; this means that for example, if the top-20 program elements have the same suspiciousness scores, we randomly select 10 out of the 20 to be the top-10. Also, in case the root cause spans more than one program element (i.e., basic block) as long as one of the program elements is in the top-10, we consider the fault localization instance to be an effective one. Recently, Parnin and Orso highlighted in their paper (Parnin and Orso 2011) that "...programmers will stop inspecting statements, and transition to traditional debugging, if they do not get promising results within the first few statements they inspect ...". That means the percentage of program elements inspected is not a suitable

evaluation metric to reflect how developers will use a fault localization tool in practice. Therefore, we use absolute rank rather than percentage rank to assess the effectiveness of fault localization instances.

Various information could be leveraged to predict if a fault localization tool is effective given a set of program execution traces. We could investigate the execution traces. If there are very few failing execution traces, then it is likely to be harder for a spectrum based fault localization tool to differentiate faulty from correct program elements. In the extreme case, when there are no test cases that expose the fault (no failing execution traces), then the output of a fault localization tool cannot be effective. We could also investigate the output of the fault localization tool. In the special case where all program elements are given the same suspiciousness score, there is a very low likelihood that the fault localization tool will be effective for those execution traces.

3 Overall Framework

The goal of our framework is to build an oracle that is able to predict if a fault localization instance is effective or not. To realize this, our framework, illustrated in Fig. 2, works on two phases: training and deployment. The training phase would output a model that is able

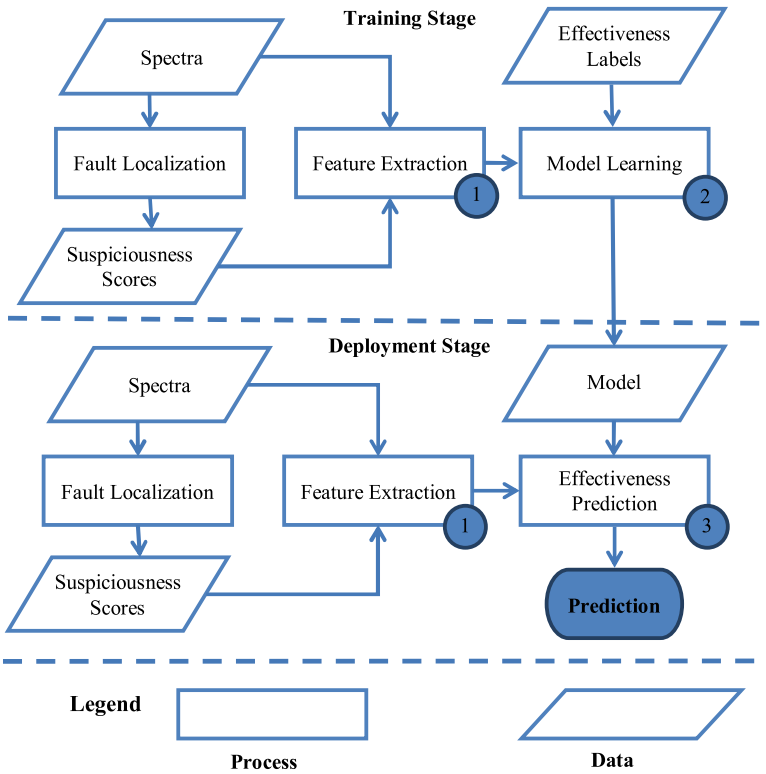

Fig. 2 Proposed framework. There are two main phases (stages) in our framework: training phase and deployment phase. Each phase consists of a number of processes. The main processes include: (1) feature extraction, (2) model learning, and (3) effectiveness prediction

to differentiate effective and ineffective fault localization instances. The deployment phase would apply this model to a number of unknown fault localization instances and output if the cases are likely to be effective or not. Let us describe these two phases in more detail.

In the training phase, we take in a set of fault localization instances. Some of these cases are effective and some others are ineffective. Each of these cases is represented by the following:

1. Program *spectra* corresponding to correct and faulty execution traces.
2. A list of *suspiciousness scores* that are assigned by the fault localization tools to the program elements.
3. An *effectiveness label*: effective (if the root cause is in the top-10) or ineffective (otherwise).

The training phase consists of two processes: feature extraction, and model learning. During feature extraction, based on a training data, we extract some feature values that shed light into some important characteristics that potentially differentiate effective from ineffective instances. In the model learning step, the feature values of each of the training instances along with the effectiveness labels are then used to build a discriminative model which is able to predict whether an unknown fault localization instance is effective or not. This discriminative model is output to the deployment stage.

The deployment stage consists of two blocks: feature extraction, and effectiveness prediction. We extract feature values from unknown instances whose labels, effective or ineffective, are to be predicted. These values are then fed to the discriminative model learned in the training phase. The model would then output a prediction.

We elaborate the feature extraction block in Section 4. The model learning and effectiveness prediction blocks are elaborated in Section 5.

4 Feature Extraction

We extract values of a number of features from input execution traces and from the outputs of a fault localization tool. Table 2 shows these features. We have in total 50 features. Fifteen of the features are extracted from input execution traces and the remaining thirty five features are extracted from the suspiciousness scores output by the tool.

The first fifteen input features capture information about program execution traces and program elements covered by these execution traces. Features T_1 to T_5 capture information on the number of execution traces available for fault localization. Too few number of traces might cause poor fault localization performance especially if there are too few failing traces. In the worst case where the number of failing traces is zero, the fault localization tool reduces to random guess. Features PE_1 to PE_4 capture the information on program elements that are covered by the execution traces. The more the number of program elements, the more difficulty a fault localization tool is likely to have as it needs to compare and differentiates more elements. With more program elements, the more likely a faulty program element to be assigned the same or lower suspiciousness scores as other program elements. Feature PE_5 captures cases where some program elements only appear in faulty but not correct executions. Intuitively, the chance for such cases to be effective is likely to be high. Feature PE_6 captures the opposite which might indicate omission errors: some program elements that should be executed are not executed. Features PE_7 to PE_{10} capture the two highest proportions of failures that passed by one program element. Intuitively, the higher the proportion of failures that passes a program element, the more likely it is the root cause.

Table 2 List of features used in this work. In total, we use 50 features

ID	Description
Input: Traces (5 Features)	
T_1	Number of traces
T_2	Number of failing traces
T_3	Number of passing traces
T_4	$T_3 - T_2$
T_5	$\frac{T_2}{T_3}$
Input: Program Elements (10 Features)	
PE_1	Number of program elements covered in the failing execution traces
PE_2	Number of program elements covered in the correct execution traces
PE_3	$PE_2 - PE_1$
PE_4	$\frac{PE_1}{PE_2}$
PE_5	Number of program elements that appear only in failing execution traces
PE_6	Number of program elements that appear only in correct execution traces
PE_7	Highest proportion of failing execution traces that pass by one program element
PE_8	Second highest proportion of failing execution traces that pass by one program element
PE_9	$PE_7 - PE_8$
PE_{10}	$\frac{PE_8}{PE_7}$
Output: Raw Scores (10 Features)	
R_1	Highest suspiciousness score
R_2	Second highest suspiciousness score
R_i	i^{th} highest suspiciousness score, where $3 \leq i \leq 10$
Output: Simple Statistics (6 Features)	
SS_1	Number of distinct suspiciousness scores in $\{R_1, \ldots, R_{10}\}$
SS_2	Mean of $\{R_1, \ldots, R_{10}\}$
SS_3	Median of $\{R_1, \ldots, R_{10}\}$
SS_4	Mode of $\{R_1, \ldots, R_{10}\}$
SS_5	Variance of $\{R_1, \ldots, R_{10}\}$
SS_6	Standard deviation of $\{R_1, \ldots, R_{10}\}$
Output: Gaps (11 Features)	
G_1	$R_1 - R_2$
G_2	$R_2 - R_3$
G_i	$R_i - R_{(i+1)}$, where $3 \leq i \leq 9$
G_{10}	$Max_{1 \leq i \leq 9}(G_i)$
G_{11}	$Min_{1 \leq i \leq 9}(G_i)$
Output: Relative Differences (8 Features)	
C_1	$\frac{(R_2 - R_{10})}{(R_1 - R_{10})}$
C_i	$\frac{(R_{(i+1)} - R_{10})}{(R_1 - R_{10})}$, where $2 \leq i \leq 8$

The next thirty five output features capture the suspiciousness scores that are output by the fault localization tool. Features R_1 to R_{10} capture the top-10 suspiciousness scores. If the suspiciousness scores are too low, intuitively it is less likely for a fault localization instance to be effective. Features SS_1 to SS_6 compute some simple statistics of the top-10

suspiciousness scores. They serve as statistical summary of the scores. Features G_1 to G_{11} and C_1 to C_8 are aimed to capture a "break" or gap in the top-10 suspiciousness scores. This "break" shows that the localization tool is able to differentiate some program elements to be significantly more suspicious than the others. That might indicate that some of the top-10 program elements are probably to be the root cause. If the fault localization tool is unable to differentiate program elements, it is less likely to be effective. In the worst case, if it is unable to distinguish all program elements, fault localization again turns into random guess.

5 Model Learning & Effectiveness Prediction

We first describe our model learning process. Next, we describe how we apply the model to effectiveness prediction.

5.1 Model Learning

As inputs to this process, we have a set of training instances with their effectiveness labels. Each of the instance is represented as 50 feature values (aka. a feature vector) produced by the feature extraction process described in Section 4. The goal of the model learning process is to convert these set of feature vectors into a discriminative model that could predict the effectiveness label of a fault localization instance whose effectiveness is unknown.

We build upon and extend a state-of-the-art classification algorithm namely Support Vector Machine (SVM) (Han and Kamber 2006). SVM has been used in many past software engineering research studies, e.g., Anvik et al. (2006), Sun et al. (2010), Maiga et al. (2012), Thung et al. (2012), and Tian et al. (2012). We first describe standard off-the-shelf SVM. We then describe our extended SVM that handles the issue of imbalanced data caused since there are more ineffective fault localization instances than effective ones.

5.1.1 Off-the-Shelf SVM

SVM solves the classification problem by looking for a linear optimal separating hyperplane, which separates data instances of one class from another (Vapnik 2000). The chosen hyperplane is called *maximum marginal hyperplane* (MMH) in which the separation between two classes are maximized. For example, consider a training dataset in form of $(\vec{x_k}, y_k)$, where $\vec{x_k}$ is the feature vector of the k^{th} training data instance. Each y_k represents class label of data instance ($y_k \in \{+1, -1\}$). The problem of searching for a separating hyperplane with maximal margin could be reduced to finding the minimal value of $\frac{1}{2}\|\vec{w}\| = \frac{1}{2}\sqrt{{w_1}^2 + \cdots + {w_n}^2}$ which satisfies the constrains: $y_k(\vec{w} \cdot \vec{x_k} + b) \geq 1 \forall k$, where $\vec{w}$ is perpendicular to the separating hyperplan, n is the number of attributes, and b is a constant number indicates position of the hyperplan in multi-dimensional space. In this study, we use SVM^{light} version 6.02[1] with linear kernel.

5.1.2 SVM^{Ext}

Imbalanced training data is one of the issues that we encounter during the course of our study. There are more ineffective than effective fault localization instances. Thus we build

[1] http://svmlight.joachims.org/

upon standard off-the-shelf SVM to address this imbalanced data problem. We call our solution SVM^{Ext}.

The pseudo-code of our proposed SVM^{Ext} is shown in Fig. 3. The algorithm takes as input a set of effective and ineffective fault localization instances - EI and II. We first check if there are more ineffective than effective localization instances (Line 1). If there are, we perform a data balancing step (Lines 2-8). We would like to duplicate effective instances that appear close to the hyperplane – these are effective instances that are close to one of the ineffective instances. In order to find these effective instances, we compute the similarity between each effective instance with each of the ineffective instances (Line 2). Each fault localization instance could be viewed as a 50-dimensional vector; each dimension is a feature and a localization instance is represented by the values of the 50 features described in Section 4. To measure the similarity between two instances we compute the Cosine similarity (Salton and McGill 1983) of their representative vectors. Consider two vectors $(a_1, \ldots, a_{50})$ and $(b_1, \ldots, b_{50})$. The Cosine similarity of these two vectors is defined as:

$$\frac{\sum_{i=1}^{50}(a_i \times b_i)}{\sqrt{\sum_{i=1}^{50}(a_i)^2} \times \sqrt{\sum_{i=1}^{50}(b_i)^2}}$$

Next, for each effective instance, we calculate its highest similarity with an ineffective instance (Line 3). We sort the effective instances based on their highest similarities with ineffective instances (Line 4). We then insert these instances from the most similar to the least similar to the collection of effective instances EI until the number of effective instances matches that of ineffective ones (Lines 5-8). We then proceed to learn a model using off-the-shelf SVM and output the resultant model (Lines 9-10).

5.2 Effectiveness Prediction

The discriminative model learned in the model learning phase would be able to predict if an unknown instance (i.e., a fault localization instance whose effectiveness is unknown) is effective or not. The unknown instance needs to be transformed to a set of feature values

Procedure SVM^{Ext}
Inputs:
EI: Effective fault localization instances
II: Ineffective fault localization instances
Output: Discriminative model
Method:
1: If $(|EI| < |II|)$
2: Let S_i^j = Similarity between $EI[i]$ (i.e., the i^{th} effective instance) with $II[j]$ (i.e., the j^{th} ineffective instance)
3: Let $M_i = Max_{j \in \{0,\ldots,|II|-1\}} S_i^j$
4: Let $MOSTSIM$ = Sorted EI (sorted in descending order of M_i)
5: Let $idx = 0$
6: While$(|EI| < |II|)$
7: Add $MOSTSIM[idx\%|MOSTSIM|]$ to EI
8: idx++
9: Let $Model$ = Model learned with off-the-shelf SVM with EI and II as training data
10: **Output** $Model$

Fig. 3 Pseudocode of SVM^{Ext} algorithm

using the feature extraction process described in Section 4. These feature values (aka. a feature vector) are then compared with the model and a prediction would be output. The feature vector is compared with the hyperplane that separates effective and ineffective training instances. Based on which side of the hyperplane the feature vector lies, the corresponding unknown instance is assigned either effective or ineffective prediction label.

6 Experiments

In this section we first describe our dataset, followed by our evaluation metrics, research questions, and results.

6.1 Dataset

We analyze 10 different programs. These include NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite (Hutchins et al. 1994). These programs have been widely used in past studies on fault localization and thus could collectively be used as a benchmark (Jones and Harrold 2005; Renieris and Reiss 2003; Liu et al. 2005; Abreu et al. 2007). Table 3 provides the details on the programs.

NanoXML is an XML parsing utility written in Java. We download NanoXML from Software Infrastructure Repository (SIR) (Do et al. 2005). SIR contains 5 variants of NanoXML: NanoXML_v1, NanoXML_v2, NanoXML_v3, NanoXML_v4, and NanoXML_v5. Each of the variants contains faulty versions except NanoXML_v4. We downloaded all 32 faulty versions of these variants. We exclude two of the faulty versions as there are no failure-inducing test cases that expose the bugs. Thus, for NanoXML, in total, we analyze 30 faulty versions. XML-Security is a digital signature and encryption library written in Java. There are 3 variants of XML-Security in SIR: XMLSec_v1, XMLSec_v2, and XMLSec_v3. For each variant, several faulty versions are provided. In total, we downloaded 52 faulty

Table 3 Dataset descriptions: name, lines of code, programming language, number of faulty versions, and number of test cases

Dataset	LOC	Language	# Faulty	# Tests
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
NanoXML v1	3,497	Java	6	214
NanoXML v2	4,007	Java	7	214
NanoXML v3	4,608	Java	9	216
NanoXML v5	4,782	Java	8	216
XML security v1	21,613	Java	6	92
XML security v2	22,318	Java	6	94
XML security v3	19,895	Java	4	84

versions from these variants; we analyze 16 of them, as there are no failure-inducing test cases that expose the other bugs. Space was used in European Space Agency and is an interpreter for Array Definition Language (ADL) written in C. All 35 faulty versions of Space downloaded from SIR are used for our experiments. For these 3 programs, in total we analyze, 81 faulty versions.

Siemens programs are originally created for a study on test coverage adequacy performed by researchers from Siemens Corporation Research (Hutchins et al. 1994). Each of the seven programs has many faulty versions derived "by seeding realistic faults" (Hutchins et al. 1994). Each faulty version contains one bug that may span more than one program element (i.e., basic block). It comes with test cases and bug free versions. Siemens programs have been used in many fault localization studies including (Jones and Harrold 2005; Renieris and Reiss 2003; Liu et al. 2005; Abreu et al. 2007). The Siemens test suite[2] include the following programs: print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. There are a total of 132 versions in the test suite. We instrumented each blocks in the versions. We exclude versions that are seeded by bugs residing in variable declarations as our instrumentation cannot reach these declarations. Thus, we exclude the following versions: version 12 of replace, versions 13, 14, 15, 36, 38 of tcas, and versions 6, 10, 19, 21 of tot_info. Versions 4 and 6 of print_token are also excluded because they are identical with the bug free version. We exclude version 9 of schedule2 as running all test cases only produces correct executions – no test case is a failure-inducing one. In total, we include 119 faulty versions from Siemens test suite for our experiment. Adding the 81 faulty versions from the 3 other programs, we have in total 200 faulty versions.

6.2 Evaluation Metrics & Experiment Settings

We evaluate the accuracy of our solution in terms of precision, recall, and F-measure. These metrics have been frequently used to evaluate various prediction engines (Han and Kamber 2006). We first define the concepts of true positives, false positives, true negatives, and false negatives:

True Positives (TP):	Number of effective fault localization instances that are predicted correctly
False Positives (FP):	Number of ineffective fault localization instances that are predicted wrongly
True Negatives (TN):	Number of ineffective fault localization instances that are predicted correctly
False Negatives (FN):	Number of effective fault localization instances that are predicted wrongly

Based on the above concepts, we can define precision, recall, and F-measure as follows:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F - Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

[2]We use the variant at: www.cc.gatech.edu/aristotle/Tools/subjects

There is often a trade-off between precision and recall. Higher precision often results in lower recall (and vice versa). To capture whether an increase in precision (or recall) outweighs a reduction in recall (or precision), F-measure is often used. F-measure is the harmonic mean of precision and recall and it combines the two measures together into a single summary measure. Both precision and recall are important. If precision is low, it means that many of fault localization instances predicted as effective are actually ineffective. Inspecting ineffective fault localization instances costs developers time and resource. Thus these false positive cases are bad. On the other hand, if recall is low, it means that many of effective fault localization instances are wrongly predicted as ineffective. These false negative cases correspond to missed opportunities; developers lose out on the opportunity to benefit from effective usages of a fault localization tool. Thus false negative cases are also bad. We believe precision and recall should be given equal weight. Thus, in this paper, we compute F-measure (i.e., F1) which is a harmonic mean of precision and recall that considers the weights of precision and recall to be the same.

We perform ten-fold cross validation to evaluate the effectiveness of our proposed approach. Ten-fold cross validation is a standard approach in data mining to estimate the accuracy of a prediction engine (Han and Kamber 2006). Its goal is to assess how the result of a prediction engine generalizes to an independent test data. In ten-fold cross validation, we divide the dataset into ten groups. To create these ten groups, we initialize ten empty buckets. We then randomly assign each fault localization instance into one of the ten buckets such that the numbers of instances in the buckets are equal or differ at most by one. At the end of the process, instances in each bucket become a group. We use nine of the groups for training and one of the groups for testing. We repeat the process 10 times using different groups as the test group. We aggregate all the results and compute the final precision, recall, and F-measure.

6.3 Research Questions

We would like to answer the following research questions. The research questions capture different aspects that measure how good our proposed approach is.

RQ1. How effective is our approach in predicting the effectiveness of a popular spectrum-based fault localization tool?

We first evaluate the accuracy of our tool in predicting the effectiveness of Tarantula which is a well-known fault localization tool.

RQ2. Could our approach be used to predict the effectiveness of other spectrum-based fault localization tools?

There are different spectrum-based fault localization tools proposed in the literature. We would like to investigate if our approach also works for these spectrum-based fault localization tools. We consider the following tools: Ochiai (Abreu et al. 2007), Information Gain (Lucia et al. 2010), $ER1^a$ (Xie et al. 2013), $ER1^b$ (Xie et al. 2013), $ER5^a$ (Xie et al. 2013), $ER5^b$ (Xie et al. 2013), $ER5^c$ (Xie et al. 2013), and DStar (Wong et al. 2014).

RQ3. How effective is our extended Support Vector Machine (SVM^{Ext}) compared with the off-the-shelf Support Vector Machine (SVM)?

To learn a discriminative model, we extend SVM to address the data imbalance issue. We would like to investigate if this extension is necessary to make our framework effective.

RQ4. How effective is our extended Support Vector Machine (SVM^{Ext}) compared with a popular solution for imbalanced learning?

We are interested to compare the performance of SVM^{Ext} with random undersampling (Japkowicz 2000; He and Garcia 2009) that is often used to handle imbalance data. Random undersampling randomly removes data instances from the majority class in order to make the number of training instances belonging to the two classes (in our case: effective and ineffective) to be the same. We denote the application of SVM algorithm with random undersampling to balance training data as SVM^{us}. We calculate and compare F-measures achieved by SVM^{Ext} and SVM^{us}.

RQ5. What are some important features that help in discriminating if a fault localization tool would be effective given a set of input traces?

We investigate which of the 50 features that we use are more dominant and thus more effective to help us achieve higher prediction accuracy. In the machine learning community, Fisher score is often used to measure how dominant or discriminative a feature is, e.g., Duda et al. (2001) and Gu et al. (2011). We compute the Fisher score of every feature as follows:

$$FS(j) = \frac{\sum_{class=1}^{\#class} (\bar{x}_j^{(class)} - \bar{x}_j)^2}{\sum_{class=1}^{\#class} (\frac{1}{n_{class}-1} \sum_{i=1}^{n_{class}} (x_{i,j}^{(class)} - \bar{x}_j^{(class)})^2)}$$

In the equation, $FS(j)$ denotes the Fisher score of the j^{th} feature. n_{class} is the numbers of data points (i.e., fault localization instances) with label $class$ (i.e., effective or ineffective). $\bar{x}_j$ denotes the average value of the j^{th} feature of all data points. $\bar{x}_j^{(class)}$ is the average value of the j^{th} feature of $class$-labeled data points. $x_{i,j}^{(class)}$ denotes the value of the j^{th} feature of the i^{th} $class$-labeled data point. Fisher scores range from 0 to 1. Features with higher Fisher scores are more discriminative, while features with Fisher scores of 0 are not discriminative.

RQ6. How sensitive is our approach to the amount of training data?

We use ten-fold cross validation to evaluate our approach. In ten-fold cross validation, we use 90 % of the data for training, and the remaining 10 % for testing. In this research question, we investigate the impact of reducing the number of training data on the accuracy of the proposed approach.

RQ7. Could data from one software program be used to train a discriminative model used to predict effectiveness of a fault localization tool on failures from other software programs?

To answer this research question, we use data from N-1 (i.e., 9) software programs to build a model. This model is then used to predict the effectiveness of a fault localization tool on the remaining one software program. We refer to this process as N-fold cross-program validation.

RQ8. How effective are various classifiers in predicting the effectiveness of a fault localization instance?

In the previous research questions, we only investigate the effectiveness of SVM^{Ext} and SVM. However, there are many other classification algorithms proposed in the literature. In this research questions, we investigate several other classifiers including J48 (Quinlan 1993), KStar (Cleary and Trigg 1995), and IBk (Aha and Kibler 1991), and see if they are as effective as SVM in predicting the effectiveness of a fault localization instance.

RQ9. How effective is our approach in predicting the effectiveness of a fault localization instance on multi-bugs setting (i.e., more than one program element are responsible for the failures)?

In the previous research questions, we only consider single-bug setting (i.e., only one program element is responsible for the failures). In this research question, we want to see if it is as effective in multi-bugs setting. We reuse the multi-bugs dataset created by Lucia et al. (2014). The multi-bugs dataset contains 173 versions each injected with 2-5 bugs. In the multi-bugs setting, we define a fault localization instance to be effective if a root cause of a bug appears in the top-10 most suspicious program elements (ties are randomly broken).

6.4 Results

In this section, we answer our research questions one at a time by performing a set of experiments.

6.4.1 RQ1: Overall Accuracy

To answer our first research question, we simply run Tarantula on the 200 faulty versions. We then predict if Tarantula is effective or not for each of the 200 faulty versions using SVM^{Ext}. We perform ten-fold cross validation and aggregate the result for the final precision, recall, and F-measure. For Tarantula, 85 of the localization instances are effective and 115 of the instances are ineffective. Thus, the data is imbalanced.

The result of our experiment is shown in Table 4. The result shows that we can achieve a precision of 54.36 %. This means that we can correctly identify *many* ineffective fault localization instances (i.e., 47 out of the 115 ineffective instances). Note that if our approach is not used, none of the ineffective fault localization instances can be identified. Thus, our approach can reduce wasted developer effort by 40.87 % (i.e., 47/115). Past studies have shown that an approach that can result even in a small reduction of wasted developer effort is useful, e.g., Jalbert and Weimer (2008). Furthermore, we can achieve a recall of 95.29 %. This means that we correctly identify *almost all* effective instances (i.e., 81 out of the 85 effective instances). F-measure, the harmonic mean of precision and recall, is often used to gauge on how effective a prediction engine is. Our F-measure is 69.23 %. Comparing with many other studies performing other prediction tasks in software engineering research literature, e.g., Seo and Kim (2012) and Shihab et al. (2012), our F-measure is comparable or higher.

Table 4 Precision, recall, and F-measure of our proposed approach in predicting effectiveness of Tarantula

Precision	54.36 %
Recall	95.29 %
F-Measure	69.23 %

6.4.2 RQ2: Different Fault Localization Tools

We also investigate if our approach could be generalized to other spectrum-based fault localization tools aside from Tarantula. We use the same set of 200 faulty versions and perform the same ten-fold cross validation using SVM^{Ext} to evaluate eight other spectrum-based fault localization tools: Ochiai (Abreu et al. 2007), Information Gain (Lucia et al. 2010), $ER1^a$ (Xie et al. 2013), $ER1^b$ (Xie et al. 2013), $ER5^a$ (Xie et al. 2013), $ER5^b$ (Xie et al. 2013), $ER5^c$ (Xie et al. 2013), and DStar (Wong et al. 2014). We run each spectrum-based fault localization tool independently. For each tool, we train a model and use the model to predict if other fault localization instances of the same tool are effective or not. Table 5 shows the precision, recall, and F-measure when we predict the effectiveness of Tarantula, Ochiai, Information Gain, $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, $ER5^c$, and DStar.

Comparing the results with those of Tarantula, we note that higher precision, recall, and F-measure can be achieved for predicting the effectiveness of Ochiai, Information Gain, $ER1^a$, $ER1^b$, and DStar. Our approach can achieve a F-measure of more than 75 % for Ochiai, Information Gain, and DStar. It can achieve a F-measure of more than 80 % for $ER1^a$ and $ER1^b$. However, the performance of our approach is lower for $ER5^a$, $ER5^b$, and $ER5^c$. Predicting the effectiveness of $ER5^a$, $ER5^b$, and $ER5^c$ is harder as their corresponding fault localization instance datasets are *highly* imbalanced. Most (i.e., 74.5 %) of the fault localization instances are ineffective.

6.4.3 RQ3: SVM^{Ext} vs. SVM

Next, we compare our extended SVM (SVM^{Ext}) with standard off-the-shelf SVM. We consider 8 scenarios depending on the target fault localization tool: Tarantula, Ochiai, Information Gain, $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, $ER5^c$, and DStar.

Tarantula. The precision, recall, and F-measure of using SVM^{Ext} and SVM for Tarantula is shown in Table 6. We also compute the relative improvement of SVM^{Ext} over SVM by the following formula:

$$Relative\ Improvement = \frac{(SVM^{Ext}\ Result - SVM\ Result)}{SVM\ Result}$$

Table 5 Precision, recall, and F-measure of our approach in predicting effectiveness of various fault localization tools

Tool	Precision	Recall	F-Measure
Tarantula	54.36 %	95.29 %	69.23 %
Ochiai	63.23 %	97.03 %	76.56 %
Information Gain	64.47 %	93.33 %	76.26 %
$ER1^a$	74.38 %	90.00 %	81.45 %
$ER1^b$	74.38 %	90.00 %	81.45 %
$ER5^a$	42.50 %	100.00 %	59.65 %
$ER5^b$	42.50 %	100.00 %	59.65 %
$ER5^c$	42.50 %	100.00 %	59.65 %
DStar	66.23 %	94.44 %	77.86 %

Table 6 Precision, recall, and F-measure of SVM^{Ext} and SVM in predicting effectiveness of Tarantula

	SVM^{Ext}	SVM	Relative Improvement
Precision	54.36 %	51.04 %	6.50 %
Recall	95.29 %	57.65 %	65.29 %
F-Measure	69.23 %	54.14 %	27.87 %

SVM^{Ext} clearly outperforms SVM with respect to precision, recall, and F-measure. We find that SVM^{Ext} outperforms SVM in terms of precision, recall, and F-Measure by 6.50 %, 65.29 %, and 27.87 %, respectively.

Ochiai. The precision, recall, and F-measure of using SVM^{Ext} and SVM for Ochiai is shown in Table 7. We find that the performance of SVM^{Ext} and SVM are the same. There is no difference in the performance of SVM^{Ext} and SVM since the fault localization instance dataset of Ochiai is much more balanced than that of Tarantula (i.e., 50.50 % of the fault localization instances are effective, and the rest are ineffective).

Information Gain. The precision, recall, and F-measure of using SVM^{Ext} and SVM for Information Gain is shown in Table 8. We find that the performance of SVM^{Ext} and SVM are the same. There is no difference in the performance of SVM^{Ext} and SVM since the fault localization instance dataset of information gain is much more balanced than that of Tarantula (i.e., 52.50 % of the fault localization instances are effective, and the rest are ineffective).

$ER1^a$. The precision, recall, and F-measure of using SVM^{Ext} and SVM for $ER1^a$ is shown in Table 9. We find that the performance of SVM^{Ext} and SVM are the same. There is no difference in the performance of SVM^{Ext} and SVM since the fault localization instance dataset of $ER1^a$ is much more balanced than that of Tarantula (i.e., 50.00 % of the fault localization instances are effective, and the rest are ineffective).

$ER1^b$. The precision, recall, and F-measure of using SVM^{Ext} and SVM for $ER1^b$ is shown in Table 10. We find that the performance of SVM^{Ext} and SVM are the same. There is no difference in the performance of SVM^{Ext} and SVM since the fault localization instance dataset of $ER1^b$ is much more balanced than that of Tarantula (i.e., 50.00 % of the fault localization instances are effective, and the rest are ineffective).

$ER5^a$. The precision, recall, and F-measure of using SVM^{Ext} and SVM for $ER5^a$ is shown in Table 11. We find that SVM is not able to predict the effectiveness of $ER5^a$ (its precision, recall, and F-measure score are all 0). Thus, SVM^{Ext} clearly outperforms SVM.

Table 7 Precision, recall, and F-measure of SVM^{Ext} and SVM in predicting effectiveness of Ochiai

	SVM^{Ext}	SVM	Relative Improvement
Precision	63.23 %	63.23 %	0 %
Recall	97.03 %	97.03 %	0 %
F-Measure	76.56 %	76.56 %	0 %

Table 8 Precision, recall, and F-measure of SVM^{Ext} and SVM in predicting effectiveness of information gain

	SVM^{Ext}	SVM	Relative Improvement
Precision	64.47 %	64.47 %	0 %
Recall	93.33 %	93.33 %	0 %
F-Measure	76.26 %	76.26 %	0 %

Note that the improvement of SVM^{Ext} over SVM is more pronounced for $ER5^a$ than for Tarantula since the fault localization instance dataset of $ER5^a$ is much more imbalanced than that of Tarantula (i.e., 25.50 % of the fault localization instances are effective, and the rest are ineffective).

$ER5^b$. The precision, recall, and F-measure of using SVM^{Ext} and SVM for $ER5^b$ is shown in Table 12. We find that SVM is not able to predict the effectiveness of $ER5^b$ (its precision, recall, and F-measure score are all 0). Thus, SVM^{Ext} clearly outperforms SVM. Note that the improvement of SVM^{Ext} over SVM is more pronounced for $ER5^b$ than for Tarantula since the fault localization instance dataset of $ER5^b$ is much more imbalanced than that of Tarantula (i.e., 25.50 % of the fault localization instances are effective, and the rest are ineffective).

$ER5^c$. The precision, recall, and F-measure of using SVM^{Ext} and SVM for $ER5^c$ is shown in Table 13. We find that SVM is not able to predict the effectiveness of $ER5^c$ (its precision, recall, and F-measure score are all 0). Thus, SVM^{Ext} clearly outperforms SVM. Note that the improvement of SVM^{Ext} over SVM is more pronounced for $ER5^c$ than for Tarantula since the fault localization instance dataset of $ER5^c$ is much more imbalanced than that of Tarantula (i.e., 25.50 % of the fault localization instances are effective, and the rest are ineffective).

DStar. The precision, recall, and F-measure of using SVM^{Ext} and SVM for DStar is shown in Table 14. We find that the performance of SVM^{Ext} outperforms SVM in terms of precision and F-measure by 1.30 % and 0.76 %, respectively (their recall scores are the same). Note that the fault localization instance dataset of DStar is much more balanced than that of Tarantula (i.e., 54 % of the fault localization instances are effective, and the rest are ineffective).

6.4.4 RQ4: SVM^{Ext} vs. SVM^{us}

We compare SVM^{Ext} with the application of SVM that uses random undersampling to handle imbalanced training data (i.e., SVM^{us}). We use the two approaches to predict the

Table 9 Precision, recall, and F-measure of SVM^{Ext} and SVM in predicting effectiveness of $ER1^a$

	SVM^{Ext}	SVM	Relative Improvement
Precision	74.38 %	74.38 %	0 %
Recall	90.00 %	90.00 %	0 %
F-Measure	81.45 %	81.45 %	0 %

Table 10 Precision, recall, and F-measure of SVM^{Ext} and SVM in predicting effectiveness of $ER1^b$

	SVM^{Ext}	SVM	Relative Improvement
Precision	74.38 %	74.38 %	0 %
Recall	90.00 %	90.00 %	0 %
F-Measure	81.45 %	81.45 %	0 %

effectiveness of 9 fault localization techniques, and compute relative improvement between SVM^{Ext} and SVM^{us} as follows:

$$Relative\ Improvement = \frac{(\text{SVM}^{Ext}\ Result - \text{SVM}^{us}\ Result)}{\text{SVM}^{us}\ Result}$$

Table 15 shows F-measures of SVM^{Ext} and SVM^{us} when predicting effectiveness of Tarantula, Ochiai, Information Gain, $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, $ER5^c$, and DStar. From the table, we have zero relative improvements for 4 out of the 9 fault localization techniques, i.e., Ochiai, Information Gain, $ER1^a$, and $ER1^b$. For these 4 techniques, we find that there are more effective fault localization instances than ineffective ones (i.e., more than 50 % of the instances are effective). On the other hand, there are 5 out of the 9 fault localization techniques that have none-zero relative improvements. They are Tarantula, $ER5^a$, $ER5^b$, $ER5^c$, and DStar where $ER5^a$ has the highest relative improvement (i.e., 8.77 %). Among the 5 techniques, only DStar has more effective instances than ineffective ones (i.e., 54 % of the instances are effective). Overall, our proposed technique for handling imbalanced training data (i.e., SVM^{Ext}) outperforms random undersampling for 5 out of 9 fault localization techniques, especially for harder cases when there is more pronounced data imbalance and the number of effective instances are fewer than the number of ineffective ones.

6.4.5 RQ5: Important Features

Next, we investigate which features are important. We use Fisher score to rank the features. We consider 9 scenarios depending on the target fault localization tool: Tarantula, Ochiai, Information Gain, $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, $ER5^c$, and DStar.

Tarantula. Table 16 shows the list of the top-10 most important features. Interestingly, we find that the top-10 features include input and output features. Both input execution traces and suspiciousness scores generated by a fault localization tool are important to predict the effectiveness of a fault localization instance.

Relative difference features, i.e., C7, C8, C6, C5, and C1, are the most discriminative (5 out of the top-10 features). These features can capture a "break" or gap in the top-10 discriminative scores. This "break" signifies that the fault localization tool is able to differentiate some program elements to be significantly more suspicious than the others. Three of

Table 11 Precision, recall, and F-measure of SVM^{Ext} and SVM in predicting effectiveness of $ER5^a$

	SVM^{Ext}	SVM	Relative Improvement
Precision	42.50 %	0 %	∞
Recall	100 %	0 %	∞
F-Measure	59.65 %	0 %	∞

Table 12 Precision, recall, and F-measure of SVM^{Ext} and SVM in predicting effectiveness of $ER5^b$

	SVM^{Ext}	SVM	Relative Improvement
Precision	42.50 %	0 %	∞
Recall	100 %	0 %	∞
F-Measure	59.65 %	0 %	∞

the top-10 features are related to program elements, i.e., PE1, PE2, and PE4. They capture the number of program elements covered in execution traces. The more program elements are covered, the harder it is to get effective fault localization as the fault localization tool needs to differentiate more program elements to find the root cause. The other two of the top-10 features are the highest suspiciousness score (R1) and the number of distinct suspiciousness scores in the top-10 scores (SS1). These are intuitively related to fault localization effectiveness: the higher a suspiciousness score is, the more likely a program element is the root cause; the more the number of distinct suspiciousness scores, the more that a fault localization tool differentiates program elements.

Ochiai. Table 16 shows the list of the top-10 most important features of Ochiai. Ignoring the ordering of the features, Ochai and Tarantula are sharing 7 important features. They are C5, C6, C7, C8, SS1, PE1, and PE2. But, different from Tarantula, the list of top-10 most important features for Ochiai also contains information of gaps between suspiciousness scores (i.e., G10) and information on the variance and standard deviation of of the suspiciousness scores (i.e., SS5 and SS6).

Information Gain. Table 16 shows the list of the top-10 most important features. The top 6 features for Information Gain also appear in the top-10 for Tarantula. For Information Gain, the 7^{th} to 10^{th} most important features do not appear in the top-10 for Tarantula. They are: T3, T1, T4, and C2. C2 is similar to other relative difference features that appear in the top-10 for Tarantula. Different from Tarantula, for Information Gain, features related to number of traces (T3, T1 and T4) are also important in predicting the effectiveness of an information-gain-based fault localization tool outputs.

$ER1^a$. Table 16 shows the list of the top-10 most important features. Half of the top-10 features for $ER1^a$ appear in the top-10 for Tarantula. The features that do not appear in the top-10 for Tarantula are: R8, R7, SS2, SS6, and R9. Different from Tarantula, for $ER1^a$, features related to raw scores (R8, R7, R9), mean of the raw scores (SS2), and standard deviation of the raw scores (SS6) are also important in predicting the effectiveness of $ER1^a$ outputs.

$ER1^b$. Table 16 shows the list of the top-10 most important features. Seven out of the top-10 features for $ER1^b$ also appear in the top-10 for Tarantula. The features that do not appear in the top-10 for Tarantula are: C4, PE6, and PE3. C4 is similar to other relative difference features that appear in the top-10 for Tarantula. PE6 and PE3 are similar to other program element features that appear in the top-10 for Tarantula.

[2]Please refer to Table 2 for the description of the features.

Table 13 Precision, recall, and F-measure of SVM^{Ext} and SVM in predicting effectiveness of $ER5^c$

	SVM^{Ext}	SVM	Relative Improvement
Precision	42.50 %	0 %	∞
Recall	100 %	0 %	∞
F-Measure	59.65 %	0 %	∞

$ER5^a$. Table 16 shows the list of the top-10 most important features. Only 2 of the top-10 features for $ER5^a$ appear in the top-10 for Tarantula – they are: PE2, and PE1. The other 8 that do not appear in the top-10 for Tarantula are: PE6, PE3, T1, T3, PE5, R10, R5, and R4. PE6, PE3, and PE5 are similar to other program element features that appear in the top-10 for Tarantula. Different from Tarantula, for $ER5^a$, features related to number of traces (T1 and T3) and raw scores (R10, R5, and R4) are also important in predicting the effectiveness of $ER5^a$ outputs.

$ER5^b$. Table 16 shows the list of the top-10 most important features. Only 2 of the top-10 features for $ER5^b$ appear in the top-10 for Tarantula – they are: PE2, and PE1. The other 8 that do not appear in the top-10 for Tarantula are: PE6, PE3, R10, R5, R4, R6, R8, and R7. PE6 and PE3 are similar to other program element features that appear in the top-10 for Tarantula. Different from Tarantula, for $ER5^b$, features related to raw scores (R10, R5, R4, R6, R8, and R7) are also important in predicting the effectiveness of $ER5^b$ outputs.

$ER5^c$. Table 16 shows the list of the top-10 most important features. Only 3 of the top-10 features for $ER5^c$ appear in the top-10 for Tarantula – they are: PE2, PE1, and SS1. The other 7 that do not appear in the top-10 for Tarantula are: PE6, PE3, T1, T3, PE5, T2, and R10. PE6, PE3, and PE5 are similar to other program element features that appear in the top-10 for Tarantula. Different from Tarantula, for $ER5^c$, features related to number of traces (T1, T3, and T2) and raw scores (R10) are also important in predicting the effectiveness of $ER5^c$ outputs.

DStar. Table 16 shows the list of the top-10 most important features. Only 1 feature of the top-10 features for DStar appears in the top-10 for Tarantula (i.e., C8). Different other fault localization techniques in our study, DStar has no important features extracted from input traces and program elements. Five out of the top-10 features are raw suspiciousness score features (i.e., R10, R9, R8, R7, and R6), and 3 out of the top-10 are simple statistics (i.e., SS2, SS5, and SS6). The other two are C8 and G10 that capture the gap and relative difference of suspiciousness scores, respectively.

Table 16 shows that top-10 most discriminative features of 9 fault localization techniques are different from one another. This is because suspiciousness scores that are output

Table 14 Precision, recall, and F-measure of SVM^{Ext} and SVM in predicting effectiveness of DStar

	SVM^{Ext}	SVM	Relative Improvement
Precision	66.23 %	65.38 %	1.30 %
Recall	94.44 %	94.44 %	0.00 %
F-Measure	77.86 %	77.27 %	0.76 %

Table 15 F-measures of SVM^{Ext} and SVM^{us}

SBFL	SVM^{Ext}	SVM^{us}	Relative Improvement
Tarantula	69.23 %	68.62 %	0.89 %
Ochiai	76.56 %	76.56 %	0 %
Information Gain	76.26 %	76.26 %	0 %
$ER1^a$	81.45 %	81.45 %	0 %
$ER1^b$	81.45 %	81.45 %	0 %
$ER5^a$	59.65 %	54.84 %	8.77 %
$ER5^b$	59.65 %	56.35 %	5.86 %
$ER5^c$	59.65 %	56.35 %	5.86 %
DStar	77.86 %	77.39 %	0.61 %

The First Column Specifies Spectrum-Based Fault Localization Techniques Whose Effectiveness is to be Predicted. The Second and Third Columns are F-measures of SVM^{Ext} and SVM^{us}, respectively. The Last Column is Relative Improvements of SVM^{Ext} over SVM^{us}

by each technique have distinct characteristics. From the formulas of the 9 techniques (see Section 2), we find that each of the techniques combines basic spectrum-based fault localization statistics listed in Table 1 in various ways. Some formulas are in the form of fractions, while some others are arithmetic expressions with plus and minus operators only. DStar involves exponentiation which results in a large number. These result in suspiciousness scores of different techniques having different characteristics. For example, Tarantula and Ochiai's suspiciousness scores are real numbers in range of [0,1]; Information Gain and DStar's suspiciousness scores are non-negative real numbers, but different from Tarantula and Ochiai, their values can be far greater than 1. Furthermore, suspiciousness scores of some techniques are real numbers while others are integers. For example, $ER1^a$, $ER5^a$, and $ER5^c$'s suspiciousness scores are integers. Hence, the differences between distinct suspiciousness scores of the 3 techniques have to be at least 1. However, for suspiciousness scores that are real numbers, the differences between two distinct suspiciousness scores can

Table 16 Important features of Tarantula, Ochiai, information gain, $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, $ER5^c$, and DStar

	Tarantula	Ochiai	IG	$ER1^a$	$ER1^b$	$ER5^a$	$ER5^b$	$ER5^c$	**D***
1	C7	C8	PE1	R8	SS1	PE2	PE2	PE2	SS6
2	C8	SS1	PE2	C7	C7	PE1	PE1	PE1	R10
3	C6	C7	C7	SS1	C6	PE6	PE6	PE6	R9
4	PE1	PE1	C8	R7	PE2	PE3	PE3	PE3	R8
5	PE2	C6	SS1	C6	PE1	T1	R10	T1	R7
6	SS1	PE2	C6	PE2	C8	T3	R5	T3	G10
7	C5	SS6	T3	SS2	C5	PE5	R4	PE5	SS5
8	C1	G10	T1	SS6	C4	R10	R6	T2	C8
9	PE4	C5	T4	R9	PE6	R5	R8	SS1	SS2
10	R1	SS5	C2	C8	PE3	R4	R7	R10	R6

"IG" and "D*" Stand for Information Gain and DStar, Respectively. List of Features is Available at Table 2

be between 0 and 1. Consequently, these diverse characteristics of output suspiciousness scores make top important features for the 9 fault localization techniques different.

Important features for fault localization techniques with similar formulas tend to be similar. For example, the formulas for $ER5^a$, $ER5^b$, and $ER5^c$ look different, but actually they are rather similar. $ER5^a$ is simply n_f^e (i.e., number of failures that execute program element e), while $ER5^b$ is a normalized n_f^e, and $ER5^c$ is an indicator function defined based on n_f^e. Thus, the top-4 features of these 3 fault localization techniques are the same.

6.4.6 RQ6: Different Amount of Training Data

In ten-fold cross validation, we use 90 % of the data for training on only 10 % for testing. To answer this research question, we vary the amount of training data from 10 % to 90 % and show the resultant precision, recall, and F-measure. We randomly pick the data that we use for training. We consider 9 scenarios depending on the target fault localization tool: Tarantula, Ochiai, Information Gain, $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, $ER5^c$, and DStar.

Tarantula. We show the result in Fig. 4. Note that as we randomly resample the 90 % data, the result is different with that of RQ1. We find that the performance of our framework does not degrade too much (F-measure $>$ 60 %) if there is sufficient data for training (30-90 %), the performance degrades substantially if there is too little training data (10-20 %).

Ochiai. We show the result in Fig. 5. Note that as we randomly resample the 90 % data, the result is different with that of RQ2. We find that the performance of our framework does not degrade too much (F-measure $>$ 60 %) if there is sufficient data for training (20-90 %), the performance degrades substantially if there is too little training data (10 %).

Information Gain. We show the result in Fig. 6. Note that as we randomly resample the 90 % data, the result is different with that of RQ2. We find that the performance drops

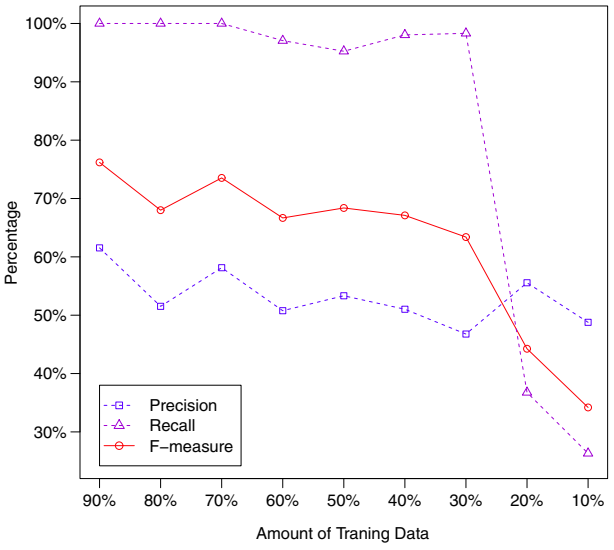

Fig. 4 Precision, recall, and F-measure of our approach in predicting effectiveness of Tarantula for various amount of training data

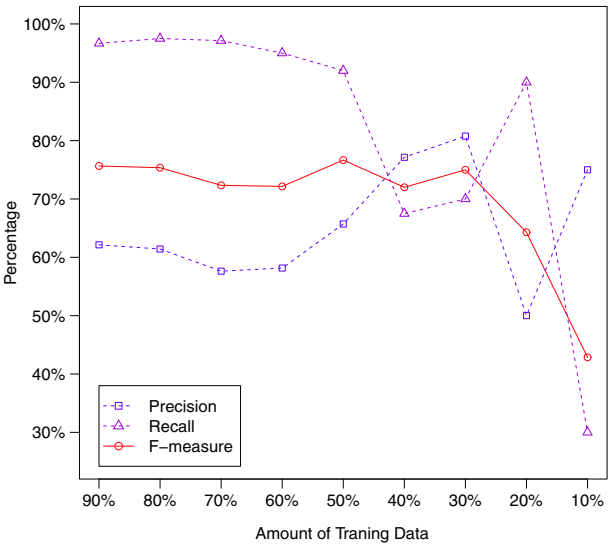

Fig. 5 Precision, recall, and F-measure of our approach in predicting effectiveness of Ochiai for various amount of training data

substantially for 60 %, 30 %, and 20 % of the training data. For the other percentages the performance does not degrade too much (F-measure $>$ 60 %).

$ER1^a$. We show the result in Fig. 7. Note that as we randomly resample the 90 % data, the result is different with that of RQ2. We find that the performance of our framework does

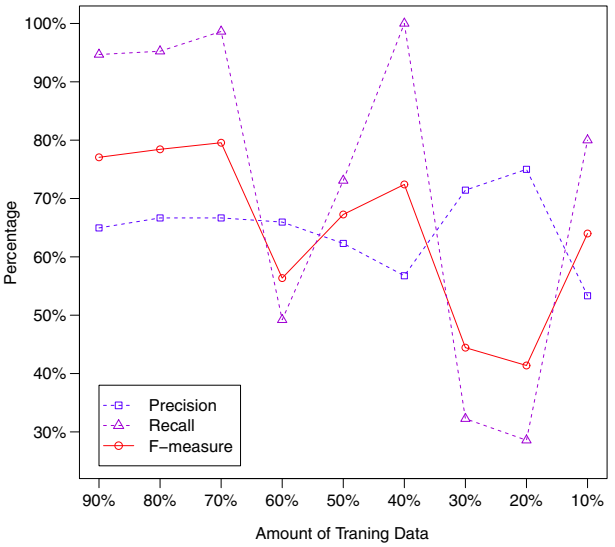

Fig. 6 Precision, recall, and F-measure of our approach in predicting effectiveness of information gain for various amount of training data

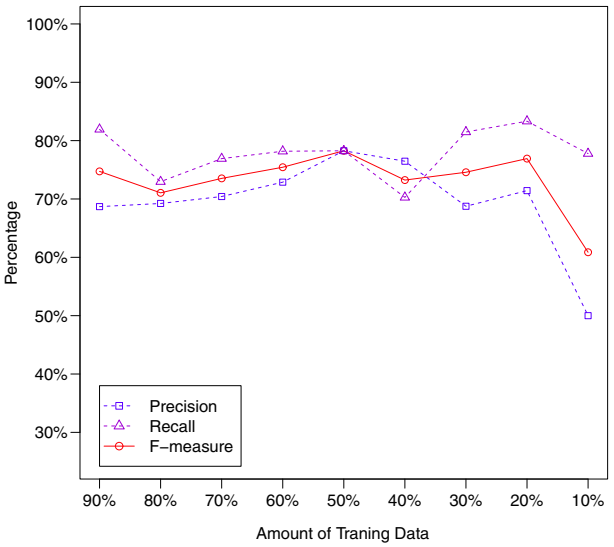

Fig. 7 Precision, recall, and F-measure of our approach in predicting effectiveness of $ER1^a$ for various amount of training data

not degrade too much (F-measure $>$ 60 %) for all percentages of the training data that we investigate (10 %-90 %).

$ER1^b$. We show the result in Fig. 8. Note that as we randomly resample the 90 % data, the result is different with that of RQ2. We find that the performance of our framework does

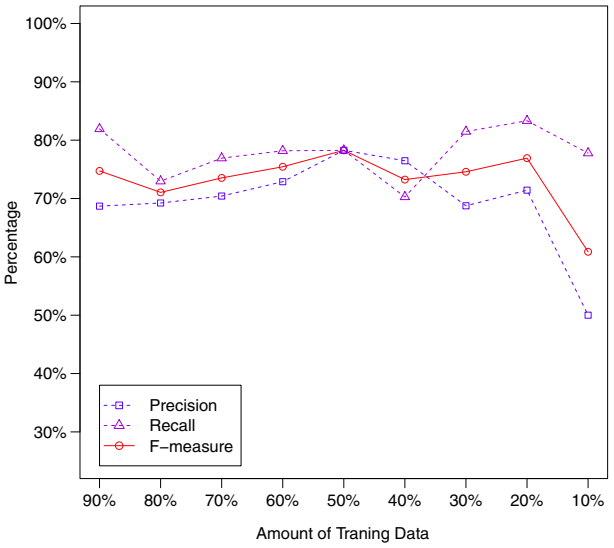

Fig. 8 Precision, recall, and F-measure of our approach in predicting effectiveness of $ER1^b$ for various amount of training data

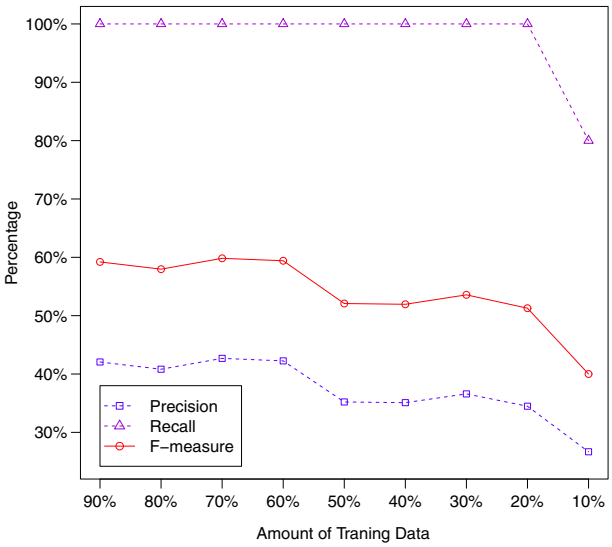

Fig. 9 Precision, recall, and F-measure of our approach in predicting effectiveness of $ER5^a$ for various amount of training data

not degrade too much (F-measure > 60 %) for all percentages of the training data that we investigate (10-90 %).

$ER5^a$. We show the result in Fig. 9. Note that as we randomly resample the 90 % data, the result is different with that of RQ2. We find that the performance of our framework does not degrade too much (F-measure > 50 %) if there is sufficient data for training (20-90 %), the performance degrades substantially if there is too little training data (10 %).

$ER5^b$. We show the result in Fig. 10. Note that as we randomly resample the 90 % data, the result is different with that of RQ2. We find that the performance of our framework does not degrade too much (F-measure > 50 %) if there is sufficient data for training (20-90 %), the performance degrades substantially if there is too little training data (10 %).

$ER5^c$. We show the result in Fig. 11. Note that as we randomly resample the 90 % data, the result is different with that of RQ2. We find that the performance of our framework does not degrade too much (F-measure > 50 %) if there is sufficient data for training (20-90 %), the performance degrades substantially if there is too little training data (10 %).

DStar. We show the result in Fig. 12. Note that as we randomly resample the 90 % data, the result is different with that of RQ2. We find that the performance drops substantially for 30 % of the training data. For the other percentages the performance does not degrade too much (F-measure > 60 %).

6.4.7 RQ7: Cross-program Setting

We perform N-fold cross-program validation to answer this research question. The results are presented in Table 17. Comparing the cross-program setting with the default setting

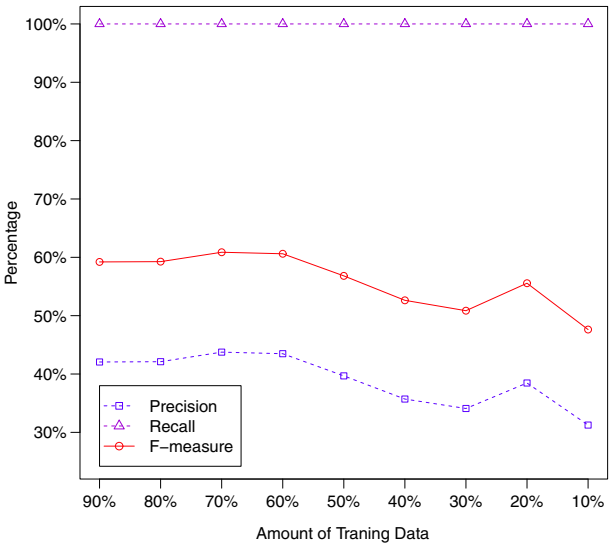

Fig. 10 Precision, recall, and F-measure of our approach in predicting effectiveness of $ER5^b$ for various amount of training data

considered in earlier RQs, there is however a modest loss in F-measure. For example, for Tarantula, the F-measure is 63.43 %, which is lower than the result for RQ1 (i.e., 69.23 %). The fact that a model learned on a set of programs will have a reduced performance when applied to a new program is well known in the literature (Zimmermann et al. 2009). This reduction is expected, but does the performance remain reasonable? The F-measures of our

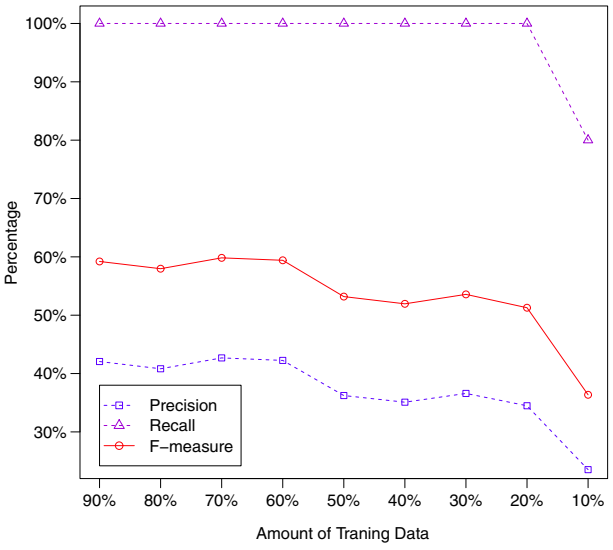

Fig. 11 Precision, recall, and F-measure of our approach in predicting effectiveness of $ER5^c$ for various amount of training data

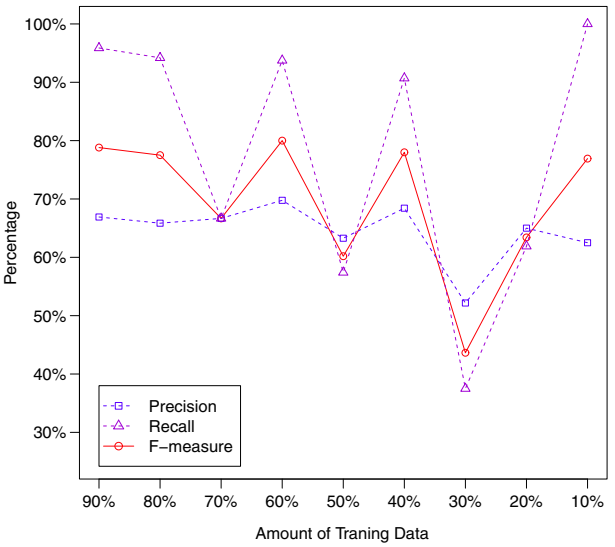

Fig. 12 Precision, recall, and F-measure of our approach in predicting effectiveness of DStar for various amount of training data

approach in predicting the effectiveness of DStar, Ochiai, Information Gain, $ER1^a$, and $ER1^b$ in the cross-project setting are all above 70 %. F-measure of 70 % or higher is often considered reasonable in the literature (Hindle et al. 2009; Kim et al. 2008; Shihab et al. 2010; Jing et al. 2014). Thus, for 5 out of the 9 fault localization techniques, our approach can perform reasonably well.

Our approach still works for the cross-program setting since we build a discriminative model using instances from n projects rather than only one project. Consequently, the model is not tuned or over fitted to one project. Thus, features that are considered important in the model are likely those that can generalize across multiple projects. Still, a project might have its own peculiar characteristics, and thus the performance of our approach in the cross-project setting is lower than when a training data from the same project is available.

Table 17 Precision, recall, and F-measure of our approach in predicting effectiveness of various fault localization tools for cross-program setting

Tool	Precision	Recall	F-Measure
Tarantula	46.4 %	100.00 %	63.43 %
Ochiai	54.89 %	100.00 %	70.88 %
Information Gain	57.07 %	100.00 %	72.66 %
$ER1^a$	54.35 %	100.00 %	70.42 %
$ER1^b$	54.35 %	100.00 %	70.42 %
$ER5^a$	29.48 %	100.00 %	45.54 %
$ER5^b$	28.81 %	100.00 %	44.74 %
$ER5^c$	28.81 %	100.00 %	44.74 %
DStar	58.70 %	100.00 %	73.97 %

Table 18 Precision, recall, and F-measure in predicting effectiveness of tarantula using various classifiers

Tool	Precision	Recall	F-Measure
SVM^{Ext}	54.36 %	95.29 %	69.23 %
SVM	51.04 %	57.65 %	54.14 %
J48	65.93 %	70.59 %	68.18 %
KStar	65.93 %	70.59 %	68.18 %
IBk	65.91 %	68.24 %	67.05 %

6.4.8 RQ8: Effectiveness of Different Classifiers

We consider 9 scenarios depending on the target fault localization tool: Tarantula, Ochiai, Information Gain, $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, $ER5^c$, and DStar.

Tarantula. The results are shown in Table 18. In terms of F-measure, which is the harmonic mean of precision and recall, we can see that the performance of SVM^{Ext} is the best followed by J48 and KStar.

Ochiai. The results are shown in Table 19. In terms of F-measure, we can see that SVM^{Ext} and SVM are the best performing classifiers followed by IBk and KStar.

Information Gain. The results are shown in Table 20. In terms of F-measure, we can see that SVM^{Ext} and SVM are the best performing classifiers followed by KStar and J48.

$ER1^a$. The results are shown in Table 21. In terms of F-measure, we can see that SVM^{Ext} and SVM are the best performing classifiers followed by J48 and IBk.

$ER1^b$. The results are shown in Table 22. In terms of F-measure, we can see that SVM^{Ext} and SVM are the best performing classifiers followed by IBk and J48.

$ER5^a$. The results are shown in Table 23. In terms of F-measure, we can see that SVM^{Ext} is the best performing classifiers followed by KStar and IBk. Note that the performance of SVM^{Ext} differs substantially from that of the worst two classifiers (SVM and J48).

$ER5^b$. The results are shown in Table 24. In terms of F-measure, we can see that SVM^{Ext} is the best performing classifiers followed by KStar and IBk. Note that the performance of SVM^{Ext} differs substantially from that of the worst two classifiers (SVM and J48).

Table 19 Precision, recall, and F-measure in predicting effectiveness of Ochiai using various classifiers

Tool	Precision	Recall	F-Measure
SVM^{Ext}	63.23 %	97.03 %	76.56 %
SVM	63.23 %	97.03 %	76.56 %
J48	64.22 %	69.31 %	66.67 %
KStar	70 %	69.31 %	69.65 %
IBk	71.43 %	74.26 %	72.82 %

Table 20 Precision, recall, and F-measure in predicting effectiveness of information gain using various classifiers

Tool	Precision	Recall	F-Measure
SVM^{Ext}	64.47 %	93.33 %	76.26 %
SVM	64.47 %	93.33 %	76.26 %
J48	68.7 %	75.24 %	71.82 %
KStar	73.53 %	71.43 %	72.46 %
IBk	71.15 %	70.48 %	70.81 %

$ER5^c$. The results are shown in Table 25. In terms of F-measure, we can see that SVM^{Ext} is the best performing classifiers followed by KStar and IBk. Note that the performance of SVM^{Ext} differs substantially from that of the worst two classifiers (SVM and J48).

6.4.9 RQ9: Multi-bugs Setting

In this research question, we investigate the performance of our approach to predict the effectiveness of 9 fault localization techniques on multi-bugs setting. Our dataset contains 173 faulty versions. Different from single-bug setting, each of the versions are injected with 2-5 bugs. That means there are more failed test cases in multi-bugs setting than single-bug setting. On average, there are 2364.13 failed test cases for a faulty version in the multi-bugs dataset. This is much larger compared to the average number of failed test cases for a faulty version in the single-bug dataset, which is only 416.87.

In multi-bugs setting, we find that 6 out of 9 fault localization techniques have effective instances account for more than 50 % of all instances. They are Tarantula, Ochiai, Information Gain, $ER1^a$, $ER1^b$, and DStar. Among the 6 techniques, Information Gain has the highest percentage of effective instances (i.e., 69.94 %). We also analyze and note that the formulas of the 6 techniques calculate suspiciousness scores by utilizing n_f^e (i.e., number of failed test cases that execute program element e) and n_s^e (i.e., number of passed test cases that execute program element e). On the other hand, for the remaining 3 techniques (i.e., $ER5^a$, $ER5^b$, and $ER5^c$), the percentages of effective instances are only 39.88 %, 39.88 %, and 39.31 %, respectively. We analyze and note that the formulas of the 3 techniques only take into account information of n_f^e, and ignore n_s^e. Hence, for cases where two program elements have the same n_f^e value, but have different n_s^e values, $ER5^a$, $ER5^b$, and $ER5^c$ still assign the same suspiciousness scores to the two elements. That explains why the 3 techniques have more ineffective instances than effective ones.

Table 21 Precision, recall, and F-measure in predicting effectiveness of $ER1^a$ using various classifiers

Tool	Precision	Recall	F-Measure
SVM^{Ext}	74.38 %	90 %	81.45 %
SVM	74.38 %	90 %	81.45 %
J48	75.23 %	82 %	78.47 %
KStar	74.29 %	78 %	76.1 %
IBk	76.92 %	80 %	78.43 %

Table 22 Precision, recall, and F-measure in predicting effectiveness of $ER1^b$ using various classifiers

Tool	Precision	Recall	F-Measure
SVM^{Ext}	74.38 %	90.00 %	81.45 %
SVM	74.38 %	90.00 %	81.45 %
J48	74.11 %	83.00 %	78.30 %
KStar	75.76 %	75.00 %	75.38 %
IBk	79.61 %	82.00 %	80.79 %

Table 27 shows the performance of our approach in multi-bugs setting. From the table, we can note that for multi-bugs setting, the results are generally good (i.e., F-measure of around 70 % or higher) for all fault localization techniques.

The fault localization formulas can be divided into two groups. The first group consists of simpler formulas namely $ER5^a$, $ER5^b$, and $ER5^c$ which are defined as n_f^e (number of failures that execute a program element e), a normalized n_f^e, and an indicator function defined on n_f^e, respectively. The other group consists of the other 6 formulas which are more complicated and consider both n_f^e and n_s^e. We find that the effectiveness of the simpler formulas can be better predicted than the effectiveness of the more complex formulas. Our approach can achieve a F-measure of 84 % for the simpler formulas, but it only achieves a F-measure of 70-75 % for the more complex formulas.

To investigate the reason why the effectiveness of the simpler formulas can be predicted with higher accuracy, we analyze and inspect the top important features for the simpler formulas in multi-bugs setting. We note that T2 (i.e., number of failing traces) and T3 (i.e., number of passing traces) are among the top important features. Interestingly, we discover that our approach, by using T2 and T3 only, and ignoring the other features, can still achieve the same F-measures as those listed in Table 27 for $ER5^a$, $ER5^b$, and $ER5^c$. For the other fault localization techniques (i.e., Tarantula, Ochiai, Information Gain, $ER1^a$, $ER1^b$, and DStar), our approach is unable to learn effective prediction model by using T2 and T3 only. That means F-measures of $ER5^a$, $ER5^b$, and $ER5^c$ are better than the other techniques in Table 27 because of the strong discrimination of T2 (i.e., number of failed traces) and T3 (i.e., number of passed traces) features for the three techniques. Upon closer inspection, we find that whenever the number of failing traces are too many (which also means the number of passed traces are too few) then it is highly likely that $ER5^a$, $ER5^b$, and $ER5^c$ instances are ineffective. When the number of failing traces are too many, it is likely that n_f^e scores of many program elements (including those that are not faults) are equally high. The simpler formulas cannot distinguish these program elements resulting in ineffective instances.

Table 23 Precision, recall, and F-measure in predicting effectiveness of $ER5^a$ using various classifiers

Tool	Precision	Recall	F-Measure
SVM^{Ext}	42.5 %	100 %	59.65 %
SVM	0.00 %	0.00 %	0.00 %
J48	66.67 %	27.45 %	38.89 %
KStar	54.17 %	50.98 %	52.53 %
IBk	54.17 %	50.98 %	52.53 %

Table 24 Precision, recall, and F-measure in predicting effectiveness of $ER5^b$ using various classifiers

Tool	Precision	Recall	F-Measure
SVM^{Ext}	42.5 %	100 %	59.65 %
SVM	0.00 %	0.00 %	0.00 %
J48	66.67 %	27.45 %	38.89 %
KStar	55.32 %	50.98 %	53.06 %
IBk	48.08 %	49.02 %	48.54 %

6.5 Threats to Validity

We consider three kinds of threats to validity: internal, external, and constructing validity. Threats to internal validity corresponds to experimenter bias. In our experiments, we use the programs that are manually instrumented by Lucia et al. (2010). Due to the manual instrumentation process, there might be some basic blocks that are missed (i.e., no instrumentation code is added for them). Threats to external validity corresponds to the generalizability of our findings. In this study, we have analyzed 10 different programs. These programs are widely studied in past fault localization studies and thus collectively they can be used as a benchmark. We have also analyzed programs written in two programming languages: C and Java. Still, more programs can be analyzed to reduce the threat further. We plan to do this in a future work. Threats to construct validity corresponds to the suitability of our metrics. We use standard metrics of precision, recall, and F-measure. These are well known metrics in data mining, machine learning, and information retrieval and have been used in many past studies in software engineering, e.g., Huang et al. (2011), Maiga et al. (2012), and Anvik et al. (2006). Thus with respect to these metrics, we believe there is little threat to construct validity. Another threat to construct validity is our definition of effective fault localization instance. In this preliminary study, we consider an instance is effective if at least one of the root cause is in the top-10 most suspicious program elements. Other definitions of effective fault localization could be considered, e.g., the root cause must be in the top-1 most suspicious program elements for an instance to be effective, etc. We leave the consideration of other definitions of effective fault localization for future work.

7 Related Work

In this section, we highlight a number of studies in *spectrum-based* fault localization which analyze program traces or their abstractions which capture the runtime behaviors of program.

Table 25 Precision, recall, and F-measure in predicting effectiveness of $ER5^c$ using various classifiers

Tool	Precision	Recall	F-Measure
SVM^{Ext}	42.5 %	100 %	59.65 %
SVM	0.00 %	0.00 %	0.00 %
J48	66.67 %	27.45 %	38.89 %
KStar	55.56 %	49.02 %	52.08 %
IBk	52.94 %	52.94 %	52.94 %

Table 26 Precision, recall, and F-measure in predicting effectiveness of DStar using various classifiers

Tool	Precision	Recall	F-Measure
SVM^{Ext}	66.23 %	94.44 %	77.86 %
SVM	65.38 %	94.44 %	77.27 %
J48	71.55 %	76.85 %	74.11 %
KStar	73.39 %	74.07 %	73.73 %
IBk	76.85 %	76.85 %	76.85 %

Many spectrum-based fault localizations studies analyze two sets of program spectra: one set corresponding to correct executions, and another set corresponding to faulty executions (Jones and Harrold 2005; Abreu et al. 2007; Zeller 2002; Liblit et al. 2003; Liu et al. 2005; Santelices et al. 2009; Cheng et al. 2009; Lo et al. 2011; Gong et al. 2012; 2012; Lucia et al. 2010; Artzi et al. 2010). Based on these inputs, these studies would typically compute likelihood of different program elements to be the root cause of the faulty executions (aka. failures). Jones and Harrold propose Tarantula that computes the suspiciousness scores of various program elements by following this intuition: program elements that are executed more frequently by faulty executions rather than correct executions are deemed to be more suspicious (Jones and Harrold 2005). Abreu et al. propose a different formula to compute suspiciousness scores (Abreu et al. 2007). They show that their proposed formula named Ochiai is able to outperform Tarantula. Zeller proposes Delta Debugging which compares a faulty execution and a correct execution and find the minimum state differences (Zeller 2002). Liblit et al. compute predicates whose true evaluation correlates with failures (Liblit et al. 2003). This work is extended by Chao et al. which propose a work, named SOBER, that considers the repeated outcomes of predicate evaluations in a program run (Liu et al. 2005). Santelices et al. use multiple program spectra to localize faults (Santelices et al. 2009). Cheng et al. propose an approach to mine a graph-based signatures, referred to as bug signatures, that differentiates correct from faulty executions (Cheng et al. 2009). Lo et al. extend the work of Cheng et al. by minimizing signatures and fusing minimized signatures to capture the context of program errors better (Lo et al. 2011). Gong et al. after that propose a test case prioritization technique to reduce the number of test cases with known oracles for fault localization (Gong et al. 2012). Gong et al. propose interactive fault localization where

Table 27 Precision, recall, and F-measure of our approach in predicting effectiveness of various fault localization tools for multi-bugs setting

Tool	Precision	Recall	F-Measure
Tarantula	70.09 %	80.39 %	74.89 %
Ochiai	78.72 %	63.25 %	70.14 %
Information Gain	84.95 %	65.29 %	73.83 %
$ER1^a$	68.81 %	80.65 %	74.26 %
$ER1^b$	68.52 %	77.08 %	72.55 %
$ER5^a$	77.11 %	92.75 %	84.21 %
$ER5^b$	77.11 %	92.75 %	84.21 %
$ER5^c$	77.11 %	94.12 %	84.77 %
DStar	73.15 %	73.15 %	73.15 %

a fault localization tool iteratively updates its recommendation as it receives feedback from end users (Gong et al. 2012). Lucia et al. investigate many association measures and adapt them for fault localization (Lucia et al. 2010). They find that Information Gain performs the best. Wang et al. employ search-based algorithms to combine various association measures and existing fault localization algorithms (Wang et al. 2011). Artzi et al. use test generation for fault localization (Artzi et al. 2010). Wong et al. propose an approach that applies radial basis function (RBF) networks for fault localization (Wong et al. 2012a). In another study, Wong et al. propose a crosstab-based statistical approach for fault localization (Wong et al. 2012b). Recently, Wong et al. propose DStar (D*) formula to localize faults and show that DStar outperforms 38 other formulas (Wong et al. 2014).

Other spectrum-based fault localizations analyze only one set of program spectra, i.e., faulty executions (Zhang et al. 2006; Gupta et al. 2005; Jeffrey et al. 2008). These techniques typically modify program runtime states systematically to localize faulty program elements. In this work, we focus on fault localization tools that compare correct and faulty executions.

8 Conclusion and Future Work

In this study, to address the unreliability of fault localization tools, we build an oracle that can predict the effectiveness of a fault localization tool on a set of execution traces. We propose 50 features that can capture interesting dimensions that potentially differentiate effective from ineffective fault localization instances. Values of these features from a training set of faulty localization instances can be used to build a discriminative model using machine learning. This model is then used to predict if unknown instances are effective or not. We have evaluated our solution on 200 faulty versions from NanoXML, XML-Security, Space, and the 7 programs in the Siemens test suite. Our solution can achieve a precision, recall, and F-measure of up to 74.38 %, 90.00 % and 81.45 %, respectively (for $ER1^a$ and $ER1^b$). We have also tested different aspects of our solution including its ability to handle cross-program setting and multi-bugs setting and the results are promising. Our study is the first study in this line of work (i.e., predicting the effectiveness of a fault localization technique). Future research can be done to reduce the number of cases where our approach is less effective. Overall, for most of the fault localization techniques studied in this work, the F-measures of our approach are above 70 %. F-measure of 70 % or higher is often considered reasonable in the literature.

As future work, we plan to improve the F-measure of our proposed approach in the following ways: We plan to perform an in-depth analysis of cases where our proposed approach is less effective and design appropriate extension to the approach. We would also like to evaluate our approach's ability to predict the effectiveness of additional fault localization techniques, e.g., Cheng et al. (2009), Wang et al. (2011), Santelices et al. (2009), and Chilimbi et al. (2009). It is also interesting to leverage other information aside from execution traces; some failures come with textual descriptions (Zhou et al. 2012), and it would be interesting to employ advanced text mining solutions (Blei et al. 2003; Wang et al. 2009) to identify whether fault localization tools would be effective on such failures.

References

Abreu R, Zoeteweij P, van Gemund AJC (2007) On the Accuracy of Spectrum-based Fault Localization. In: TAICPART-MUTATION

Aha D, Kibler D (1991) Instance-based learning algorithms. Mach. Learn. 6:37–66
Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: ICSE, pp 361–370
Artzi S, Dolby J, Tip F, Pistoia M (2010) Directed test generation for effective fault localization. In: ISSTA
Beizer B (1990) Software Testing Techniques, 2nd edn. International Thomson Computer Press, Boston
Blei D, Ng A, Jordan M (2003) Latent Dirichlet allocation. J Mach Learn Res 3:993–1022
Cheng H, Lo D, Zhou Y, Wang X, Yan X (2009) Identifying bug signatures using discriminative graph mining. In: ISSTA
Chilimbi T, Liblit B, Mehra K, Nori A, Vaswani K (2009) HOLMES: Effective statistical debugging via efficient path profiling. In: ICSE
Cleary JG, Trigg LE (1995) K*: An instance-based learner using an entropic distance measure. In: 12th International Conference on Machine Learning, pp 108–114
Do H, Elbaum SG, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir Softw Eng 10(4):405–435
Duda R, Hart P, Stork D (2001) Pattern Classification. Wiley-Interscience Publication
Geng L, Hamilton H (2006) Interestingness measures for data mining: A survey. In: ACM Computing Surveys
Gong L, Lo D, Jiang L, Zhang H (2012) Diversity maximization speedup for fault localization. In: ASE, pp 30–39
Gu Q, Li Z, Han J (2011) Generalized fisher score for feature selection. In: UAI, pp 266–273
Gupta N, He H, Zhang X, Gupta R (2005) Locating faulty code using failure-inducing chops. In: ASE, pp 263–272
Han J, Kamber M (2006) Data Mining Concepts and Techniques, 2nd edn. Morgan Kaufmann
Harrold M, Rothermel G, Sayre K, Wu R, Yi L (2000) An empirical investigation of the relationship between spectra differences and regression faults. Software Testing, Verification and Reliability
He H, Garcia EA (2009) Learning from imbalanced data. IEEE Trans Knowl Data Eng 21(9):1263–1284
Hindle A, Germán DM, Godfrey MW, Holt RC (2009) Automatic classication of large changes into maintenance categories. In: ICPC, pp 30–39
Huang L, Ng V, Persing I, Geng R, Bai X, Tian J (2011) Autoodc: Automated generation of orthogonal defect classifications. In: ASE
Hutchins M, Foster H, Goradia T, Ostrand T (1994) Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proceedings of ICSE, pp 191–200
Jalbert N, Weimer W (2008) Automated duplicate detection for bug tracking systems. In: DSN, pp 52–61
Japkowicz N (2000) The class imbalance problem: Significance and strategies. In: Proceedings of the 2000 International Conference on Artificial Intelligence ICAI, pp 111–117
Jeffrey D, Gupta N, Gupta R (2008) Fault localization using value replacement. In: ISSTA
Jing XY, Ying S, Zhang ZW, Wu SS, Liu J (2014) Dictionary learning based software defect prediction. In: ICSE, pp 414–423
Jones J, Harrold M (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: ASE
Kim S, Jr EJW, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Trans Softw Eng 34(2):181–196
Le TDB, Thung F, Lo D (2013) Theory and practice, do they match? a case with spectrum-based fault localization. In: ICSM
Gong L, Lo D, Jiang L, Zhang H (2012) Interactive fault localization leveraging simple user feedback. In: ICSM, pp 67–76
Liblit B, Aiken A, Zheng AX, Jordan MI (2003) Bug isolation via remote program sampling. In: PLDI, pp 141–154
Liu C, Yan X, Fei L, Han J, Midkiff SP (2005) SOBER: Statistical model-based bug localization. In: ESEC/FSE
Lo D, Cheng H, Wang X (2011) Bug signature minimization and fusion. In: HASE, pp 340–347
Lucia LD, Jiang L, Budi A (2010) Comprehensive evaluation of association measures for fault localization. In: ICSM
Lucia LoD, Jiang L, Thung F, Budi A (2014) Extended comprehensive study of association measures for fault localization. Journal of Software: Evolution and Process
Maiga A, Ali N, Bhattacharya N, Sabane A, Guéhéneuc YG, Antoniol G, Aïmeur E (2012) Support vector machines for anti-pattern detection. In: ASE
Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: Dwyer MB, Tip F (eds) ISSTA. ACM, pp 199–209
Quinlan R (1993) C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, San Mateo
Renieris M, Reiss S (2003) Fault localization with nearest neighbor queries. In: ASE, pp 141–154

Reps T, Ball T, Das M, Larus J (1997) The use of program profiling for software maintenance with applications to the year 2000 problem. In: ESEC/FSE
Salton G, McGill M (1983) Introduction to Modern Information Retrieval. McGraw-Hill
Santelices R, Jones J, Yu Y, Harrold M (2009) Lightweight fault-localization using multiple coverage types. In: ICSE
Seo H, Kim S (2012) Predicting recurring crash stacks. In: ASE, pp 180–189
Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, ichi Matsumoto K (2010) Predicting re-opened bugs: A case study on the eclipse project. In: WCRE, pp 249–258
Shihab E, Ihara A, Kamei Y, Ibrahim W, Ohira M, Adams B, Hassan AE, Matsumoto K (2012) Studying re-opened bugs in open source software. Empirical Software Engineering
Sun C, Lo D, Wang X, Jiang J, Khoo SC (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: ICSE (1)
Tassey G (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology Planning Report:02–32002
Thung F, Lo D, Jiang L (2012) Automatic defect categorization. In: WCRE
Tian Y, Sun C, Lo D (2012) Improved duplicate bug report identification. In: CSMR, pp 385–390
Vapnik V (2000) The Nature of Statistical Learning Theory, 2nd edn. Springer-Verlag
Wang S, Lo D, Jiang L (2011) Search-based fault localization. In: ASE
Wang X, Lo D, Jiang J, Zhang L, Mei H (2009) Extracting paraphrases of technical terms from noisy parallel software corpora. In: ACL/IJCNLP
Wong WE, Debroy V, Golden R, Xu X, Thuraisingham BM (2012a) Effective software fault localization using an rbf neural network. IEEE Trans Reliab 61(1):149–169
Wong WE, Debroy V, Xu D (2012b) Towards better fault localization: A crosstab-based statistical approach. IEEE Trans Syst Man Cybern Part C 42(3):378–396
Wong WE, Debroy V, Gao R, Li Y (2014) The dstar method for effective software fault localization. IEEE Trans Reliab 63(1):290–308
Xie X, Chen T, Kuo FC, Xu B (2013) A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. TOSEM
Zeller A (2002) Isolating cause-effect chains from computer programs. In: FSE, pp 1–10
Zhang X, Gupta N, Gupta R (2006) Locating faults through automated predicate switching. In: ICSE
Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: ICSE
Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, pp 91–100