Learning to Combine Multiple Ranking Metrics for Fault Localization

Jifeng Xuan, Martin Monperrus

Learning to Combine Multiple Ranking Metrics for Fault Localization

Jifeng Xuan
INRIA Lille - Nord Europe
Lille, France
jifeng.xuan@inria.fr

Martin Monperrus
University of Lille & INRIA
Lille, France
martin.monperrus@univ-lille1.fr

***Abstract*—Fault localization is an inevitable step in software debugging. Spectrum-based fault localization applies a ranking metric to identify faulty source code. Existing empirical studies on fault localization show that there is no optimal ranking metric for all the faults in practice. In this paper, we propose MULTRIC, a learning-based approach to combining multiple ranking metrics for effective fault localization. In MULTRIC, a suspiciousness score of a program entity is a combination of existing ranking metrics. MULTRIC consists two major phases: learning and ranking. Based on training faults, MULTRIC builds a ranking model by learning from pairs of faulty and non-faulty source code. When a new fault appear, MULTRIC computes the final ranking with the learned model. Experiments are conducted on 5386 seeded faults in ten open-source Java programs. We empirically compare MULTRIC against four widely-studied metrics and three recently-proposed metrics. Our experimental results show that MULTRIC localizes faults more effectively than state-of-art metrics, such as Tarantula, Ochiai, and Ample.**

I. INTRODUCTION

Debugging is an expensive task in software development. *Fault localization* eases debugging by automatically analyzing bugs to find out the root cause – a specific location in source code – of the problem. *Spectrum-based fault localization* (also called *coverage-based fault localization*) is a class of fault localization approaches, which leverages the execution traces of test cases to predict the likelihood – called *suspiciousness scores* – of source code locations to be faulty. In spectrum-based fault localization, a faulty program is instrumented for collecting the running traces; then a *ranking metric* is applied to compute suspiciousness scores for *program entities* (such as methods [25], [24], basic blocks [31], and statements [12], [2]). The most suspicious entities are given to a developer for further analysis [21]. The developer manually analyzes the source code, from the most suspicious location and downward, to confirm the inferred root cause of the bug.

The ideal fault localization ranking metric would always rank the faulty source code entity at the top, by giving highest suspiciousness score. However, there is no such metric, and in practice, empirical results [18], [1], [16] have shown that metrics are more or less effective at giving discriminating suspiciousness scores to the faulty locations. The motivation of this paper is to decrease the effort (e.g., labor and time cost) of developers in fault localization activities by proposing a better way to rank program entities.

The family of ranking metrics for spectrum-based fault localization approaches is large and diverse (e.g., Tarantula [12], Ochiai [2], Jaccard [2], and Ample [4]). Most of these metrics are manually and analytically designed based on assumptions on programs, test cases, and their relationship with faults [16]. To our knowledge, only the work by Wang et al. [27] considers the combination of multiple ranking metrics. They propose search-based algorithms to form such a combination in fault localization. In this paper, we propose the combination approach of multiple ranking metrics based on machine learning. Our idea is to leverage the diversity of ranking metrics by automatically combining multiple metrics.

We propose MULTRIC, a learning-based approach that combines multiple fault localization ranking metrics. MULTRIC consists in two major phases: learning and ranking. In the learning phase, a ranking model is learned based on the ranks of program entities in training faulty programs. When a new fault happens, this is the ranking phase, in which the model computes weights for combining metrics and generating the final ranking. In other words, MULTRIC determines the final ranking of program entities based on a combination of various suspiciousness scores.

Fault localization cannot be transferred into a general classification problem because the ranks of program entities are hard to label as classes in machine learning. To solve this problem, we use a learning-to-rank method. Instead of classes in classification, a learning-to-rank method models orders between faulty entities and non-faulty entities and optimizes the model to satisfy those orderings. In other words, the ranking of program entities is converted to indicate whether an actual faulty one ranks above a non-faulty one. The learning-to-rank method is used to to combine 25 existing ranking metrics.

Experiments are conducted on 5386 seeded faults in ten open-source Java programs. We empirically compare MULTRIC against four widely-studied ranking metrics (Tarantula [12], Ochiai [2], Jaccard [2], and Ample [4]) and three recently-proposed ranking metrics (Naish1 [18], Naish2 [18], and GP13 [33]). Experimental results show that MULTRIC can localize faults more effectively than the ranking metrics.

We make the following major contributions in this paper.

1. To our knowledge, this is the first approach that automatically learns a suspiciousness model by combining multiple existing ranking metrics.

2. We empirically evaluate MULTRIC on 5386 seeded faults in ten open-source Java programs to evaluate our approach.

3. We compare the effectiveness of MULTRIC against seven

Program entity	Test case									Spectrum				Suspiciousness	
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	e_f	e_p	n_f	n_p	Ochiai	Ample
$entity_1$	•			•	•			•		1	3	4	1	0.55	0.22
faulty $entity_2$		•	•	•		•	•		•	3	3	2	1	0.15	0.55
$entity_3$		•		•	•		•	•		3	2	2	2	0.10	0.60
Pass or fail	P	F	P	F	P	P	F	P	P						

Fig. 1. Example of fault localization on a program excerpt with three program entities and nine test cases. A • indicates that a test case executes an entity. Three out of nine test cases are failing because entity $entity_2$ contains a fault. Two sampled ranking metrics, Ochiai and Ample, are used for fault localization. Both metrics cannot pinpoint the faulty entity at the top of fault localization ranking.

ranking metrics (four typical ones and three recent ones). In particular, MULTRIC beats metrics that are theoretically optimal but under some unpractical assumptions [18].

The remainder of this paper is organized as follows. In Section II, we describe the background and motivation. In Section III, we present our approach, MULTRIC. In Section IV, we list experimental design and research questions. In Section V, we show the experimental results. In Sections VI and VII, we list the threats to validity and the related work. In Section VIII, we conclude and present the future work.

II. BACKGROUND AND MOTIVATION

We introduce the background of spectrum-based fault localization and the motivation of combining multiple ranking metrics in our work.

A. Spectrum-Based Fault Localization

A program spectrum records the information related to the execution of program entities (when and which entity is executed). A program entity is a unit of source code, e.g., a class, a method, or a statement. Spectrum-based fault localization (also called coverage-based fault localization [21], [25]) is a family of techniques for ranking faulty source code entities based on spectra. In practice, spectrum-based fault localization is used to speed up the debugging process and quickly finding the root cause of failing test cases [12], [1]. Recently, fault localization has been used in the context of automatic software repair repair [14].

A spectrum of a particular program entity is a tuple of four values. Formally, the spectrum is defined as (e_f, e_p, n_f, n_p), where e_f and e_p denote the numbers of failing and passing test cases that execute the program entity while n_f and n_p denote the numbers of failing and passing test cases that do not execute the program entity under consideration. Spectra can be collected based on instrumentation of program entities. Fig. 1 illustrates a program composed of three entities and nine test cases and shows its program spectrum.

A spectrum can be converted into a suspiciousness score via a ranking metric. A *ranking metric* [18] (also a *risk evaluation formula* [33], [29] or a *similarity coefficient* [2]) is a numeric function that calculates a suspiciousness score for each program entity. Then, all program entities are ranked according to the suspiciousness score. A ranking metric is formally defined as a function of a spectrum, i.e., $susp = f(e_f, e_p, n_f, n_p)$. For instance, Tarantula by Jones et al. [12], is a widely-used ranking metric. . Tarantula measures the ratio of failing test cases and passing test cases, and has initially been developed for visualizing and localizing faults in C programs (see Table I for details).

B. Motivation

Most of spectrum-based ranking metrics are designed based on an analytical approach. No metric is systematically better than all the others. In Table 1, we take two ranking metrics, Ochiai and Ample, as examples to motivate of our work. Ochiai by Abreu et al. [2], is one of the state-of-art ranking metric in fault localization. This metric models the similarity between test cases that execute an entity and test cases that fail. Ample by Dallmeier et al. [4], is another well-known metric, which differentiates the ratio of failing test cases and the ratio of passing test cases. Empirical results [18], [33] show that both Ochiai and Ample are effective in fault localization. Ochiai and Ample are defined as Ochiai $= \frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}}$ and Ample $= |\frac{e_f}{e_f+n_f} - \frac{e_p}{e_p+n_p}|$, respectively.

We apply Ochiai and Ample to the example in Fig. 1. Among three entities, Ochiai ranks $entity_1$ as the top while Ample ranks the $entity_3$ as the top. However, both Ochiai and Ample have not localized the actual fault in the first rank. An intuitive question is raised: Is there any way to combine the suspicious scores to rank $entity_2$ as the top?

A potential solution to combine the suspiciousness scores is formed as $susp = a \cdot \text{Ochiai} + b \cdot \text{Ample}$. First, we assume that both a and b are two real numbers and that $a + b = 1$, where $a \geq 0$ and $b \geq 0$. Then if we expect $entity_2$ to be always ranked at the top, both $0.55a + 0.22b < 0.15a + 0.55b$ and $0.10a + 0.60b < 0.15a + 0.55b$ should be satisfied. In this example, no real values of a and b can be obtained. Considering those two ranking metrics, there is no fixed weights for this example.

In this paper, we propose a more sophisticated kind of solution, and assume that a and b are two functions (i.e., weighting functions) that takes program entities as input and outputs a weight. In the example in Fig. 1, a simple solution is as follows,

$$a = \begin{cases} 1 & \text{if Ochiai} \geq 0.15 \\ 0 & \text{if Ochiai} < 0.15 \end{cases} \text{ and } b = \begin{cases} 1 & \text{if Ample} \geq 0.55 \\ 0 & \text{if Ample} < 0.55 \end{cases}.$$

Under this assumption, the suspiciousness score of $entity_2$ is 0.70 while the scores of $entity_1$ and $entity_3$ are 0.55 and 0.60, respectively. Then $entity_2$ is correctly ranked at the top of the list. This paper presents an approach to learn those weighting functions.

III. PROPOSED APPROACH

We propose MULTRIC, a machine learning approach to combining MULTiple existing ranking meTRICs for fault localization. In this section, we first introduce the framework of MULTRIC. Second, we present the way of extracting multiple suspiciousness scores of program entities. Third, we show how to learn the model from training program entities; fourth, we show how to predict fault locations with the learned model; finally, technical details of the learning algorithm in MULTRIC are given.

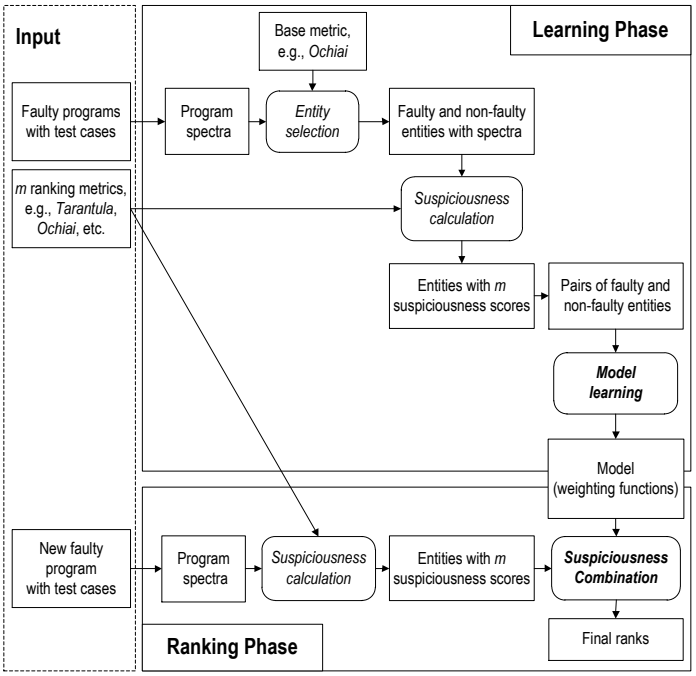

Fig. 2. Conceptual framework of MULTRIC. This framework takes faulty programs with test cases and a set of known ranking metrics as input; the output is the final ranking of program entities.

A. Framework

Our approach, MULTRIC, is a learning-based approach for fault localization. In contrast to designing ranking metrics in existing approaches, MULTRIC combines weights of existing ranking metrics to form final rankings. To build the model, MULTRIC learns weights from the spectra of faulty and non-faulty program entities. Weights of existing metrics in MULTRIC are tuned by minimizing unfit orders of faulty and non-faulty entities. In our work, we use functions to model the weights. Details will be shown in Section III-C. Informally, for a program under test, the model of MULTRIC is built to rank faulty entities above all the non-faulty ones.

Fig. 2 illustrates the overall structure of MULTRIC. This structure consists of two major phases: learning and ranking. As most approaches in spectrum-based fault localization, the input of MULTRIC is faulty programs with test cases. In addition, MULTRIC considers ranking metrics. In contrast to a single ranking metric like Tarantula, MULTRIC combines multiple ranking metrics in a weighted model instead.

In the *learning phase*, the training data is composed of faulty programs, whose program spectra are collected by executed the test cases (including at least one failing test case). Given a base fault localization metric (e.g. Ochiai), all the entities in this program are ranked. We select a subset of entities with their spectra as training data. For all program entities in the training data, we extract features, i.e., the suspiciousness scores that are calculated with a set of existing metrics. Then, all pairs of faulty entity and non-faulty entity are extracted, given that a faulty entity should always be ranked above a non-faulty one. All the pairs in the training data are then used to tune the weights of the ranking metrics to be combined.

TABLE I. THE 25 RANKING METRICS FROM THE LITERATURE THAT ARE COMBINED IN MULTRIC.

Ranking metric	Definition	Ranking metric	Definition
Tarantula	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$	Ochiai	$\frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}}$
Jaccard	$\frac{e_f}{e_f+e_p+n_f}$	Ample	$\lvert\frac{e_f}{e_f+n_f}-\frac{e_p}{e_p+n_p}\rvert$
RussellRao	$\frac{e_f}{e_f+e_p+n_f+n_p}$	Hamann	$\frac{e_f+n_p-e_p-n_f}{e_f+e_p+n_f+n_p}$
SϕrensenDice	$\frac{2e_f}{2e_f+e_p+n_f}$	Dice	$\frac{2e_f}{e_f+e_p+n_f}$
Kulczynski1	$\frac{e_f}{n_f+e_p}$	Kulczynski2	$\frac{1}{2}(\frac{e_f}{e_f+n_f}+\frac{e_f}{e_f+e_p})$
SimpleMatching	$\frac{e_f+n_p}{e_f+e_p+n_f+n_p}$	Sokal	$\frac{2e_f+2n_p}{2e_f+2n_p+n_f+e_p}$
M1	$\frac{e_f+n_p}{n_f+e_p}$	M2	$\frac{e_f}{e_f+n_p+2n_f+2e_p}$
RogersTanimoto	$\frac{e_f+n_p}{e_f+n_p+2n_f+2e_p}$	Goodman	$\frac{2e_f-n_f-e_p}{2e_f+n_f+e_p}$
Hamming	e_f+n_p	Euclid	$\sqrt{e_f+n_p}$
Overlap	$\frac{e_f}{\min(e_f,e_p,n_f)}$	Anderberg	$\frac{e_f}{e_f+2e_p+2n_f}$
Ochiai2	$\frac{e_f n_p}{\sqrt{(e_f+e_p)(n_f+n_p)(e_f+n_p)(e_p+n_f)}}$	Zoltar	$\frac{e_f}{e_f+e_p+n_f+\frac{10000 n_f e_p}{e_f}}$
Wong1	e_f	Wong2	e_f-e_p
Wong3	e_f-h, where $h=\begin{cases}e_p & \text{if } e_p\le 2\\ 2+0.1(e_p-2) & \text{if } 2<e_p\le 10\\ 2.8+0.01(e_p-10) & \text{if } e_p>10\end{cases}$		

In the *ranking phase*, the spectra of a new faulty program are collected. For each entity in this program, the same set of known ranking metrics in the learning phase is applied. Based on those metrics and the learned weights of the model, the final suspiciousness scores of these entities are computed. These scores are formed as the final fault localization result.

Application Scenario. In a long-term software project, developers and testers find and fix faults, and the information on faults is usually available in bug repositories and version control systems. Those past faults are used to extract training data; then based on the training data, MULTRIC learns the ranking model for fault localization. When a new fault is reported, MULTRIC is used to rank the source code entities in order to ease debugging. This has no overhead for developers, compared to spectrum based fault localization approaches, like Tarantula or Ochiai. The whole process of MULTRIC is run automatically, and the collection and execution of past faults can automatically be executed on continuous integration systems.

B. Suspiciousness in Ranking Metrics

The final suspiciousness score of MULTRIC is a combination of suspiciousness by multiple existing metrics. In this section, we describe the metrics in use and the method of combining suspiciousness with weights. Weights of ranking metrics are assigned during the learning phase (Section III-C) and are applied in the ranking phase (Section III-D).

We use 25 existing ranking metrics [1], [18], [33], [31] in our ranking model. Table I lists the details of these 25 metrics. Suspiciousness scores in all these ranking metrics are calculated based on the spectra of (e_f, e_p, n_f, n_p). In MULTRIC, suspiciousness scores characterize program entities. Hence, suspiciousness scores can be called *features* in a machine learning terminology.

In MULTRIC, we add a weighting function to each existing ranking metric to form the final suspiciousness score. Formally,

given a set $\mathcal{K}$ of known ranking metrics, the suspiciousness score of a program entity x in MULTRIC, is defined as follows,

$$\text{Multric}(x) = \sum_{k \in \mathcal{K}} weight_k(x) \cdot susp_k(x) \quad (1)$$

where $susp_k(x)$ denotes the suspiciousness score by a ranking metric $k \in \mathcal{K}$ and $weight_k(x)$ denotes the weight of the ranking metric k. Similar to the example in Section II-B, $weight_k(x)$ can be expressed with a real number or a function. In our work, we define $weight_k(x)$ with a function to build a refined ranking model,

$$weight_k(x) = \sum_{i=1}^{i \leq m_k} w_{ki} \cdot r_{ki}(x) \quad (2)$$

where w_{ki} is a positive number, $r_{ki}(x)$ is a binary function that returns only 1 or 0, and m_k is the times of tuning the weight for the metric k before the algorithm converges. Thus, $weight_k(x)$ is a sum of numeric weights for $r_{ki}(x)$. In general, a function $r_{ki}(x)$ is a binary classifier in machine learning, which can be conducted in various forms [23]. In our work, we use a simple definition as follows,

$$r_{ki}(x) = \|susp_k(x) \geq \theta_{ki}\| \quad (3)$$

where θ_{ki} is a learned constant as the threshold of the binary decision in $r_{ki}(x)$ and $\|\cdot\|$ is a binary operator. $\|\cdot\| = 1$ if $\cdot$ is true and $\|\cdot\| = 0$ if $\cdot$ is false. Hence, in our work, we learn $weight_k(x)$ for each known ranking metric $k \in \mathcal{K}$ from training data.

C. Learning the Model from Training Faults

We train the model of MULTRIC as follows. Informally, the model learns the orders between faulty entities and non-faulty entities: a faulty entity should always have a higher suspiciousness score than a non-faulty one. We call an ordered pair of faulty and non-faulty program entities a "training datum". Based on these ordered pairs, we train the ranking model with a learning-to-rank algorithm, which comes from information retrieval [15]. In this section, we describe how to form the training data for MULTRIC. Details of the learning-to-rank algorithm will be presented in Section III-E.

We collect training data from known faulty programs. As mentioned above, our data are in a form of pairs, which distinguish faulty and non-faulty program entities in one program. Formally, given a faulty program t, $\langle x_t^+, x_t^- \rangle$ is defined as an ordered pair of a faulty entity x_t^+ and a non-faulty entity x_t^-. This pair indicates an order that x_t^+ should be ranked above x_t^-.

Algorithm 1 shows the details of the learning process. Ideally, all the pairs can be extracted from each faulty program. However, this leads to a large amount of training data, which is time-consuming and hard to be applied in practice. In our approach, we only consider a subset of pairs using a specific non-random strategy, i.e., a neighborhood strategy.

The *neighborhood strategy* selects pairs of entities that are close based on a given initial ranking. The intuition is as follows: if two program entities are close in an initial ranking, they have similar suspsiciousness scores reflecting similar spectra, and are hard to be differentiated. This is where the weighting functions can make the difference compared to a base metric. In this neighborhood strategy, a neighbor is defined as one of the nearest non-faulty entities in an initial ranking. For example, we can extract three nearest non-faulty entities above a faulty entity and another three nearest ones below the faulty entity to form six neighbors.

Input:
$\mathcal{T}$, set of faulty programs with test cases;
$\mathcal{K}$, set of known ranking metrics, including Tarantula, Ochiai, etc;
β, base metric for ranking program entities;
γ, number of neighbors above or below faulty entities.
Output:
Ranking model
foreach *faulty program* $t \in \mathcal{T}$ **do**
  Collect spectra for an entity set S_t in program t;
  Generate ranks for each entity $x_t \in S_t$ with β;
  Add a faulty entity $x_t^+ \in S_t$ to a set $\mathcal{X}_t^+$;
  Add each non-faulty entity x_t^- of 2γ neighbors (above or below x_t^+) to $\mathcal{X}_t^-$;
end
Extract suspiciousness for all $x_t^+ \in \mathcal{X}_t^+$ with all $k \in \mathcal{K}$;
Extract suspiciousness for all $x_t^- \in \mathcal{X}_t^-$ with all $k \in \mathcal{K}$;
foreach *faulty program* $t \in \mathcal{T}$ **do**
  Form all the ordered pairs $\langle x_t^+, x_t^- \rangle$ and add to $\mathcal{T}$;
end
Build a learning-to-rank model with ordered pairs in $\mathcal{T}$;

Algorithm 1: Learning a ranking model for fault localization based on known ranking metrics.

We select neighbors above a faulty entity because we expect the faulty one is ranked above its neighbors; on the other hand, we select neighbors below a faulty entity because these neighbors should be kept below the faulty one. Based on this targeted selection of entities, we form faulty and non-faulty entities into an ordered pair. Finally, all the ordered pairs are used to learn the ranking model. To determine the initial ranking, we use one known metric to generate suspiciousness scores, e.g., Tarantula, Ochiai, or etc.

Let us now discuss two implementation details, which are omitted in the process of Algorithm 1 for sake of space. First, for each program, we only keep the one entity if two entities have the same spectrum. Especially, if a faulty entity and a non-faulty entity have the same spectrum, we only keep the faulty one. This implementation may omit some of non-faulty entities but can reduce the ambiguity for the learning algorithm. Second, we normalize suspiciousness scores from 0 to 1 (both inclusive) for each ranking metric. That is, given a ranking metric k, the normalized suspiciousness is defined as $norm\_susp_k = \frac{susp_k - min_k}{max_k - min_k}$, where min_k and max_k denote the minimum and the maximum for the metric k in the selected faulty and non-faulty entities. We conduct this normalization because some of ranking metrics may generate negative suspiciousness scores (e.g., Goodman and Wong2 in Table I); such negative scores add the complexity of learning algorithms.

D. Ranking Program Entities with the Model

In Section III-C, we learn a ranking model of MULTRIC from training faults. In this section, we present how to generate new suspiciousness scores with the learned model. As shown in Fig. 2, given a new faulty program, MULTRIC first collects program spectra for all the entities. Then existing metrics in Table I are used to calculate suspiciousness scores. Combined suspiciousness scores with their weights in the learned model, MULTRIC can calculate the final suspiciousness scores for all the entities. Finally, according these scores, we rank entities as the final result of fault localization. Similar to the learning phase, suspiciousness scores for each metric $k \in \mathcal{K}$ are normalized with min_k and max_k in the training data.

E. Algorithm: Learning to Rank

In MULTRIC, we train our ranking model with a learning-to-rank algorithm. In information retrieval, learning to rank is a family of algorithms, which re-rank documents with a learnable model using features of candidate documents in a query [15].

Learning to rank addresses the ranks of documents instead of labels of documents in general supervised leaning. Existing work in learning to rank can be divided into three categories: pointwise approaches, pairwise approaches, and listwise approaches. As their names suggest, these three categories of approaches rank documents based on the ranks of single points, pairs, and lists, respectively [26].

In our work, we adapt learning-to-rank techniques to re-rank program entities to address fault localization. Given a faulty program, the goal of fault localization is to rank faulty entities above non-faulty ones. Besides the order of faulty and non-faulty ones in a program, there is no further comparison among program entities. Thus, we apply pairwise approaches in our work.

We learn the model in MULTRIC with pairs of faulty and non-faulty program entities. The goal of learning the model is to determine the parameter values in the weighting structure in (1), including w_{ki} in (2) and θ_{ki} in (3) for a ranking metric k in the ith trial. Since our goal in MULTRIC is to rank faulty entities as the top, we define the loss function in learning to rank as the number of wrongly rankings,

$$loss = \sum_{\langle x_t^+, x_t^- \rangle} \|\text{Multric}(x_t^+) \leq \text{Multric}(x_t^-)\| \qquad (4)$$

Then the goal of learning the model in MULTRIC is converted into minimizing the loss function, i.e., minimizing the number of wrongly orders of faulty and non-faulty entities.

Many pairwise learning-to-rank approaches can solve this problem, e.g., RankBoost [8], RankNet [3], and FRank [26]. In this paper, we apply RankBoost, a classic and efficient approach based on AdaBoost [22], to learn the parameter values in the model. Algorithm details of RankBoost can be found in [8]. In this paper, we do not discuss the impact by applying different learning-to-rank algorithms.

IV. EXPERIMENTAL DESIGN

This section describes the experimental settings (including protocol, subject programs, and implementation) and the research questions in our work.

A. Protocol

We evaluate our approach by analyzing the performance of fault localization over a large number of seeded faults. In this paper, we consider object-oriented programs and choose methods as program entities following [25] and [21].

We use the absolute wasted effort of localizing a faulty method to examine the effectiveness of ranking metrics. Given a faulty subject program and a ranking metric, all the methods are ranked according to their suspiciousness scores. Then the wasted effort is defined as the rank of the actual faulty method. If more than one methods have the same suspiciousness scores as the faulty method, the wasted effort is the average ranks of all the methods with such suspiciousness scores. That is, given a set S of candidate methods, the wasted effort is formally defined as

$$effort = |\{susp(x) > susp(x^*)\}| + |\{susp(x) = susp(x^*)\}|/2 + 1/2$$

where $x \in S$ is any candidate method, x^* is the actual faulty method, and $|\cdot|$ calculates the size of a set. The value of the wasted effort is from 1 to $|S|$ (both inclusive).

The MULTRIC model is trained on faulty programs. For each subject program, we randomly select 30% of faults to form the training data and use the rest 70% of faults as new faults for evaluation. We use the average absolute wasted effort on these 70% of faults for all the ranking metrics, including MULTRIC, Tarantula, Ochiai, etc. To avoid the bias of random selection, we conduct 30 runs of the same experiment. In MULTRIC, the set $\mathcal{K}$ of known ranking metrics is composed of all 25 ranking metrics listed in Table I; the base metric β is set as Ochiai; the number of neighbors γ is set as 10 (i.e., 10 neighbors above a faulty entity and 10 neighbors below). Experiments in Sections V-C and V-D will further discuss the sensitivity of β and γ on the effectiveness of MULTRIC.

B. Subject Programs

We evaluate our approach on 5386 faults in ten Java open-source subject programs. These subject programs and their faults come from Steimann et al. [25] and are publicly accesible[1]. Table II shows the details of these subject programs, including the program version, the number of methods, the number of methods under test, the number of test cases, and the number of faults. For each subject program, faults are seeded with six mutation operators: negating conditions, replacing constants, deleting statements, inversing operators, assigning null values, and returning null values. To form all the faults in experiments, each mutation operator is run repeatedly and 100 faults are randomly selected. Thus, up to 600 faults are seeded based on the program mutation techniques (for some subject programs, a mutation operator generates less than 100 faults because the search space for mutation is insufficient). As shown in Table II, all the ten subject programs have over 350 faults.

[1] http://www.feu.de/ps/prjs/EzUnit/eval/ISSTA13.

TABLE II. DESCRIPTIVE STATISTICS OF OUR DATASET OF TEN SUBJECT PROGRAMS AND THEIR FAULTS

Subject program	# Methods	# Methods under test	# Test cases	# Faults
Daikon 4.6.4	14387	1936	157	352
Eventbus 1.4	859	338	91	577
Jaxen 1.1.5	1689	961	695	600
Jester 1.37b	378	152	64	411
JExel 1.0.0b13	242	150	335	537
JParsec 2.0	1011	893	510	598
AcCodec 1.3†	265	229	188	543
AcLang 3.0†	5373	2075	1666	599
Draw2d 3.4.2†	3231	878	89	570
HtmlParser 1.6†	1925	785	600	599
Total	29360	8397	4395	5386

† Full names of last four subject programs are Apache Commons Codec, Apache Commons Lang, Eclipse Draw2d, and HTML Parser, respectively.

C. Implementation

Our approach is implemented in Java 1.7. Experiments are conducted on a workstation with an Intel Xeon 2.6.7 CPU. The ranking algorithm in MULTRIC is implemented with an open source tool, RankLib[2]. In our implementation, the whole process of MULTRIC in Fig. 2 is run automatically.

D. Research Questions

We empirically investigate the following four Research Questions (RQs).

RQ1. How effective is our approach, MULTRIC, compared with state-of-art spectrum-based ranking metrics?

This questions investigates the most important criterion of fault localization: the effectiveness in terms of the wasted effort (as defined in IV-A). We compare our approach against four widely-studied ranking metrics (Tarantula, Ochiai, Jaccard, and Ample) and three recent-proposed ranking metrics (Naish1, Naish2, and GP13).

RQ2. How many existing ranking metrics are involved in MULTRIC?

In MULTRIC, we combine multiple existing ranking metrics to form the final ranking of suspicious entities. Not all ranking metrics are actually used in MULTRIC. We empirical examine the number of ranking metrics that are used after the learning phase. Moreover, we present which ranking metrics are used most frequently. The answer to RQ2 gives useful information on the selection of ranking metrics to fault localization practitioners.

RQ3. Which base metric is the most effective for learning the model of MULTRIC?

As explained above and shown in Fig. 2, MULTRIC uses a base metric to compute an initial ranking of entities in faulty programs. We investigate which metric is the most effective for learning the model of MULTRIC.

[2]RankLib 2.1, http://sourceforge.net/p/lemur/wiki/RankLib/.

TABLE III. NEW COMPETITORS OF MULTRIC: THREE RECENT PROPOSED RANKING METRICS

Ranking metric	Definition
Naish1 [18]	$\begin{cases} -1 & \text{if } e_f > 0 \\ n_p & \text{if } e_f = 0 \end{cases}$
Naish2 [18]	$e_f - \frac{e_p}{e_p+n_p+1}$
GP13 [33]	$e_f(1 + \frac{1}{2e_p+e_f})$

RQ4. What is the impact of the number of neighbors for learning the model in MULTRIC?

In Section III-C, we describe the data selection of faulty and non-faulty program entities. The non-faulty entities are selected based on a neighborhood strategy. We empirically study the impact of the number of neighbors, both on the wasted effort and the running time.

V. EXPERIMENTAL RESULTS

In this section, we apply the experimental protocol described in Section IV to answer four research questions. The experiments involve 5386 faults in ten subject programs.

A. RQ1. Effectiveness

To evaluate the effectiveness of fault localization, we compare the absolute wasted effort of MULTRIC with four widely-studied ranking metrics, Tarantula, Ochiai, Jaccard, and Ample (in Table I), and three recent proposed ranking metrics, including Naish1, Naish2, and GP13 (in Table III). Naish1 and Naish2 by Naish et al. [18] (respectively called Op1 and Op2 in [33]) are two formulas, which are proved as the optimal under theoretical assumptions [18]. The optimality denotes there is no higher suspiciousness score in other ranking metrics. Yoo [33] reports that Naish1 and Naish2 are strong ranking metrics in practice although not always optimal in practice. GP13 by Yoo [33] is an automatically generated ranking metric based on Genetic Programming (GP), and reported as the best ranking metric among all the GP-based ones. All of Naish1, Naish2, and GP13 have been evaluated on C programs. The results we present in this section are for Java programs.

Table IV gives the median, the average, and the standard deviation of the wasted effort over 30 runs for each subject program. Table IV shows that MULTRIC achieves the lowest wasted effort in nine out of ten subject programs among the metrics under comparison. An exception is Jester, where Naish1 gives the best result (the lowest effort). One possible reason is that the average wasted effort by most of the metrics on Jester is the lowest among the ten subject programs. That indicates that original results by Ochiai, Ample, or Naish1 are already good and hard to improve. On Eventbus, Jaxen, JExel, JParsec, AcLang, and HtmlParser, the performance of MULTRIC is over 50% better than that of Tarantula; on Jaxen and AcLang, the effort of MULTRIC is over 50% better than that of Ochiai. We also see that the standard deviation (stdev) is low, meaning that the results of MULTRIC are stable.

If we compare only the four typical ranking metrics (Tarantula, Ochiai, Jaccard, and Ample), Ochiai obtains the best results on six out of ten subject programs while Ample obtains the best on the other four. This experiment gives replication

TABLE IV. COMPARISON OF THE WASTED EFFORT ON TEN SUBJECT PROGRAMS. LOW EFFORT OF A RANKING METRIC INDICATES HIGH EFFECTIVENESS. MULTRIC IMPROVES FAULT LOCALIZATION ON MOST SUBJECT PROGRAMS.

Subject program		MULTRIC	Tarantula	Ochiai	Jaccard	Ample	Naish1	Naish2	GP13
Daikon	Median	**92.06**	132.61	129.12	129.20	143.26	147.93	160.90	131.49
	Average	**92.47**	132.62	128.97	129.04	142.76	147.05	159.94	131.47
	Stdev	4.50	7.04	6.80	8.82	6.81	8.13	7.19	6.95
Eventbus	Median	**6.39**	16.21	6.82	7.08	8.54	45.54	13.19	14.03
	Average	**6.44**	16.03	6.85	7.11	8.58	45.33	13.16	13.98
	Stdev	0.49	0.91	0.33	0.33	0.48	1.98	0.42	0.84
Jaxen	Median	**8.62**	50.76	19.65	28.59	52.69	12.12	59.33	19.62
	Average	**8.59**	50.56	19.60	28.63	52.25	12.26	59.07	19.54
	Stdev	0.57	1.98	1.16	1.38	3.06	1.40	1.18	1.36
Jester	Median	3.18	4.87	3.53	3.70	3.28	**3.09**	6.68	4.03
	Average	3.20	4.86	3.52	3.67	3.26	**3.08**	6.70	4.06
	Stdev	0.11	0.20	0.10	0.12	0.09	0.08	0.17	0.16
JExel	Median	**6.74**	15.51	9.92	10.71	9.67	7.35	14.34	10.92
	Average	**6.58**	15.44	9.87	10.74	9.57	7.17	14.28	10.84
	Stdev	0.47	0.66	0.59	0.63	0.68	0.57	0.57	0.57
JParsec	Median	**3.95**	14.02	4.77	5.52	5.68	22.90	13.35	9.19
	Average	**3.92**	14.14	4.71	5.43	5.64	22.64	13.46	9.11
	Stdev	0.38	0.88	0.33	0.42	0.52	2.44	0.62	0.58
AcCodec	Median	**3.33**	6.47	3.94	4.12	3.58	3.72	4.96	5.29
	Average	**3.33**	6.46	3.96	4.14	3.56	3.71	4.94	5.24
	Stdev	0.15	0.20	0.17	0.18	0.15	0.24	0.19	0.15
AcLang	Median	**1.83**	5.58	4.33	4.39	5.52	18.95	5.87	5.21
	Average	**1.84**	4.89	3.49	3.56	4.26	18.28	5.06	4.47
	Stdev	0.10	1.23	1.24	1.24	1.85	4.28	1.23	1.22
Draw2d	Median	**23.43**	31.46	26.05	26.33	25.96	34.38	48.57	27.62
	Average	**23.32**	31.61	26.12	26.39	26.19	33.70	48.65	27.69
	Stdev	1.06	1.59	1.49	1.51	1.52	2.37	1.32	1.47
HtmlParser	Median	**6.43**	21.25	7.15	8.58	12.25	20.75	48.92	13.70
	Average	**6.48**	21.29	7.17	8.55	11.95	21.03	48.96	13.62
	Stdev	0.54	1.15	0.36	0.45	1.15	1.95	0.90	0.73

evidence that Ochiai and Ample are strong ranking metrics on Java programs. GP13 achieves good results; this replication validates the effectiveness of its formula [33].

Let us now focus on Naish1 and Naish2, which have been claimed to be optimal [18]. They are outperformed by MULTRIC on nine subject programs, by GP13 on five subject programs and by Ochiai on three of them. A possible reason for the difference between the theoretical and empirical results is that faulty programs in practice may not follow the same assumptions.

For example, one assumption for the optimality of Naish1 and Naish2 is that test cases execute program paths in a uniform distribution (all the paths have the same execution probability during testing) [18]. This assumption is not satisfied in practice. Another assumption is that a failing test case executes at least one faulty position in source code [25], [29]. This assumption may not be satisfied by some faults. Taking Jaxen and HtmlParser as examples, a failing test case does not execute a faulty position on $8/600$ faults in Jaxen and $27/599$ faults in HtmlParser.

Answer to RQ1. Our approach MULTRIC obtains the best fault localization performance on nine out of ten subject programs in the experiments. Such results show that MULTRIC is very effective in fault localization. Meanwhile, our experiments on ten Java programs also show that Ochiai, Ample, Naish1, and GP13 give good results.

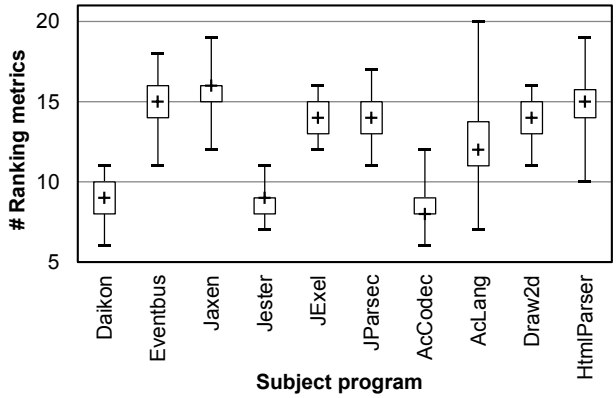

Fig. 3. A study of the number of known ranking metrics used in MULTRIC after learning (with weights over zero). A box plot shows the minimum, the first quartile, the median, the third quartile, and the maximum (no result is considered as an outlier).

B. RQ2. Features

As explained in Section III-B, we add weights to 25 known ranking metrics for computing the final suspiciousness scores. If a ranking metric is weighted to zero, the metric is not used at all in the final fault localization. Using our experimental data, we empirical analyze how many ranking metrics are actually used in MULTRIC.

Fig. 3 presents the box plot of the number of used ranking metrics (with weights over zero) over 30 individual runs. On three out of ten subject programs, Daikon, Jester, and AcCodec, MULTRIC uses a small proportion of ranking metrics, e.g. 8/25 ranking metrics (median) for AcCodec. The maximum median is for Jaxen (16/25 ranking metrics). This shows that MULTRIC indeed discards many ranking metrics at training time (based on the median, over one third of ranking metrics are discarded.) . Considering the stability for the number of ranking metrics, Jester is the most stable one, which ranges only from 7 to 11 metrics; AcLang is on the other extreme: the number of used ranking metrics varies from 7 to 20.

For each subject program, we record the selection of each ranking metric over the 30 runs. Table V presents the most selected ranking metrics in ten subject programs. As shown in Table V, several ranking metrics appear in all the 30 runs for each subject program, for example, two ranking metrics (Ample and Wong3) in AcCodec and eight ranking metrics (Kulczynski1, Kulczynski2, etc.) in Eventbus. Ample is the only ranking metric which is selected over all runs, Zoltar appears in all runs but one (for AcCodec).

The results in Table V is practical. In general, the running time of MULTRIC depends on the number of ranking metrics during the model training. A small set of ranking metrics results in low running time. Thus, for a new subject program or when computational resources are limited, we can select ranking metrics according to these empirical results. For example, Ample should be the default choice since it is the most used ranking metrics in Table V.

TABLE V. THE MOST USEFUL RANKING METRICS. MULTRIC SELECTS THESE METRICS SINCE THEY ARE USED IN MOST OF THE RUNS. RANKING METRICS THAT ARE USED IN ALL RUNS FOR A GIVEN SUBJECT PROGRAM ARE MARKED IN GREY.

Subject program	Top-1	#	Top-2	#	Top-3	#	Top-4	#	Top-5	#	Top-6	#	Top-7	#	Top-8	#	Top-9	#	Top-10	#
Daikon	Zoltar	30	Ample	30	Wong3	30	Tarantula	28	Hamming	27	Euclid	27	Wong2	26	Ochiai	24	M1	17	RogersTanimoto	16
Eventbus	Kulczynski1	30	Kulczynski2	30	M2	30	Zoltar	30	Ample	30	Wong2	30	Wong3	30	Ochiai2	30	Tarantula	28	RussellRao	28
Jaxen	Kulczynski1	30	Kulczynski2	30	M2	30	Overlap	30	Zoltar	30	Ample	30	Sokal	29	Euclid	29	Wong2	28	Hamming	26
Jester	Kulczynski1	30	M2	30	Overlap	30	Zoltar	30	Ample	30	Wong3	30	Ochiai	26	Ochiai2	20	RogersTanimoto	16	Wong2	12
JExel	Tarantula	30	Kulczynski2	30	Sokal	30	M1	30	Overlap	30	Zoltar	30	Ample	30	M2	29	Hamann	28	Kulczynski1	26
JParsec	Tarantula	30	Kulczynski2	30	M2	30	Overlap	30	Zoltar	30	Ample	30	Wong2	26	Kulczynski1	25	RogersTanimoto	22	Wong3	22
AcCodec	Ample	30	Wong3	30	Zoltar	29	M2	27	Wong2	23	RogersTanimoto	15	Overlap	15	RussellRao	14	Kulczynski1	14	Kulczynski2	12
AcLang	Zoltar	30	Ample	30	Wong3	30	Kulczynski1	28	Wong2	28	RogersTanimoto	23	Kulczynski2	21	Ochiai	19	Overlap	16	Hamming	15
Draw2d	Tarantula	30	Kulczynski1	30	M2	30	Overlap	30	Zoltar	30	Ample	30	Wong3	30	Ochiai	29	Hamming	29	Wong2	27
HtmlParser	Kulczynski1	30	Kulczynski2	30	M2	30	Overlap	30	Zoltar	30	Ample	30	M1	29	Wong2	27	Wong3	26	RogersTanimoto	26

Answer to RQ2. The number of ranking metrics used in MULTRIC varies with subject programs. The median of the number of ranking metrics in each subject program ranges from 8 to 16. A list of frequently used ranking metrics is provided and shows that Ample and Zoltar are good ranking metrics for MULTRIC according to our empirical results.

C. RQ3. Base Metric

In Algorithm 1, we initialize a ranking of program entities with a base metric β. In the experiments of Section V-A, we manually set this ranking metric to Ochiai, one of state-of-art ranking metrics. In this section, we examine the impact of different base metrics on the fault localization effectiveness.

We consider two subject programs, Jaxen and HtmlParser. We compute the results with using six ranking metrics as initialization (Tarantula, Ochiai, Jaccard, Ample, M2, and Wong3 in Table I). The first four ranking metrics are the same as those in Table IV; the other two, M2 [18] and Wong3 [28], are widely-studied in fault localization. Fig. 4 presents the comparison as a bar chart. Each value is the average wasted effort over 30 runs (a lower value is better). For the subject program Jaxen, Ochiai is the best base metric and provides the lowest wasted effort. For subject program HtmlParser, Ample is the best base metric and Ochiai is the second. We also find that the wasted effort does not change much when using other ranking as initialization, and that our approach is not sensitive to this parameter. However, it is valuable to select a good ranking metric such as Ochiai, in order to helps MULTRIC for the model learning.

Answer to RQ3. Ochiai and Ample are good choices as a base metric. An effective base metric is helpful to maximize the effectiveness of MULTRIC.

D. RQ4. Number of Neighbors

In Section III-C, we propose a neighborhood strategy to select non-faulty program entities for each faulty entity. We select the γ neighbors above or below a faulty entity. That is, if $\gamma = 5$, at most 10 neighbors are selected (sometimes there is less than 5 neighbors above or below the faulty one). Intuitively, a large number of neighbors leads to high running time, i.e., the time cost for training and applying the model. But the relationship between the number of neighbors and the wasted effort is not clear. In this section, we explore the relationship between the wasted effort and the running time for different numbers of neighbors. Note that the running time includes both the time of training the model and the time of applying the model to new faulty programs.

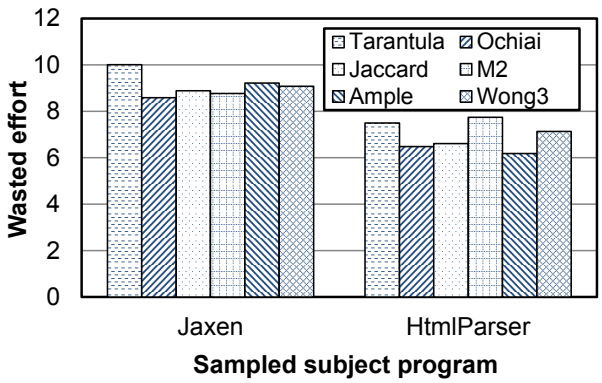

Fig. 4. A study of the impact of the base metrics used for determining the initial ranking of program entities. The approach is slightly sensitive to this tuning parameter.

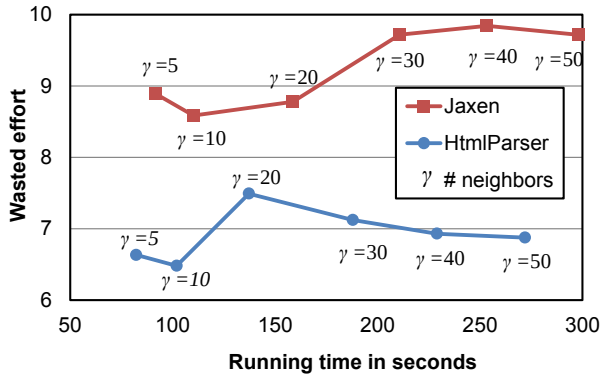

Fig. 5. A study of the impact of the number of neighbors γ on the wasted effort and the running time of MULTRIC. For low values of γ ($\gamma \leq 20$ for Jaxen and $\gamma \leq 10$ for HtmlParser), there is a tradeoff between effectiveness and performance.

We again consider two subject programs, Jaxen and HtmlParser and vary the value of γ between 5 and 50. Each dot of Fig. 5 gives the running time in seconds (x-axis) and the wasted effort (y-axis). For the two subject programs, a line joins the corresponding dots. With respect to the wasted effort, the best choice is to set γ to 10. For Jaxen, $\gamma = 20$ takes slightly more computation time than $\gamma = 5$ and there is a tradeoff between the wasted effort and the running time. Interestingly, it is never helpful to consider values of $\gamma > 20$ for Jaxen and $\gamma > 10$ for HtmlParser. Beyond such thresholds, both the effectiveness and the running time are worse and there is no tradeoff anymore.

Answer to RQ4. The number of neighbors in MULTRIC impacts the wasted effort and the running time of fault localization. A good choice is to set $\gamma = 10$, which corresponds to selecting up to 20 non-faulty program entities for each faulty program entities.

VI. THREATS TO VALIDITY

This paper proposes a learning-based approach to combining existing ranking metrics for fault localization. We discuss threats to the validity of our work in three categories.

In our work, we choose methods as the granularity of fault localization rather than statements. Existing work by Jones et al. [12], Abreu et al. [2], and Naish et al. [18] investigates fault localization with statements in C programs while Xu et al. [31] addresses the basic blocks in Java programs. In our work, we follow Steimann et al. [24], [25] use methods as the program entities. Given the program spectra for statements, our work can be easily to adapt to fault localization on statements without further modification.

In Section III-C, we learn a ranking model to recommend suspicious positions for new faults. We employ a classic learning-to-rank algorithm, RankBoost [8], to build the learnable model. However, other learning-to-rank algorithms, such as RankNet [3] or FRank [26], can be also applied to MULTRIC. In experiments, we have not examined the results of other algorithms. This may leads to a threat for the bias of a single algorithm. In MULTRIC, other learning-to-rank algorithms can be directly applied instead of RankBoost. Experimental results should be further studied in the future.

In Section IV, we evaluate MULTRIC on over 5000 faults in ten subject programs. Training faults for building the model and new faults for validating the model are both selected from the same subject program. That is, we only consider intra-subject-program faults rather than inter-subject-program faults in our experiments. In ideal scenarios, inter-subject-program fault localization is more applicable than intra-subject-program localization. However, limited by learning techniques, an effective learning model across projects is hard to build. A potential solution is transfer learning [20], which is a recent field for cross-project or cross-domain learning problems. Thus, in our work, we propose MULTRIC to build intra-subject-program models. For a long-term subject program, a large amount of faults accumulate in daily development and testing. This provides the scenario of intra-subject-program fault localization. We can apply MULTRIC to build the learnable model on past faults and to rank faulty source code for new faults.

VII. RELATED WORK

To our knowledge, this paper is the first work, which learns from multiple ranking metrics to improve fault localization. We list the related work as follows.

Jones et al. [12] propose the first well-known ranking metric, Tarantula, in spectrum-based fault localization. Abreu et al. [2] design Ochiai and Jaccard, which are also widely-studied and state-of-art ranking metrics. Jeffrey et al. [11] develop a value replacement technique by examining covered statements in failing test cases. Naish et al. [18] empirically study effectiveness of existing ranking metrics on C programs and propose Naish1 and Naish2 as two optimal ranking metrics in theory.

Xie et al. [29], [30] summarize existing ranking metrics and theoretically divide ranking metrics into equivalent groups. Lo et al. [16] propose a comprehensive study on the relationship among multiple existing ranking metrics and show that there is no best single ranking metrics in practice. Steimann et al. [25] investigate the threats to the validity in spectrum-based fault localization. This work analyzes empirical results on method-based fault localization in ten Java open-source programs.

Existing work has considered leveraging training data for fault localization. Feng & Gupta [7] propose a Bayesian network approach to modeling the dependency from source code to testing results. In this work, an error flow graph is built from randomly selected training sets to collect dependency data. Yoo [33] designs a genetic programming approach to organizing the spectrum of source code to automatically form the ranking metric. In this approach, six out of 30 generated ranking metrics have high effectiveness, including the best one, GP13.

Debroy & Wong [5] combine three existing ranking metrics (Tarantula, Ochiai, and Wong3) and localize faults based on the consensus of ranking metrics. The most related work to our paper is search-based fault localization by Wang et al. [27]. They propose search-based methods, including simulated annealing and genetic algorithms, to assign numeric weights to ranking metrics. Their work evaluates the search-base methods on 7 C program in SIR[3]. In contrast to above work [27], our MULTRIC is a combination of existing ranking metrics via learning-to-rank techniques. We model weights of ranking metrics with functions and learn the weights via pairs of faulty and non-faulty program entities. Such weighting strategy leads MULTRIC to high effectiveness. Meanwhile, experiments in our work use over 5000 Java faults while experiments in [7], [33], [27] use less than 130 C faults.

Various aspects have been studied in fault localization. Gong et al. [10] propose a framework of interactive fault localization, which leverages simple user feedback to improve the accuracy. Nguyen et al. [19] design a database-aware approach, SQLook, for the scenario of fault localization in dynamic web applications. Zhang et al. [35] propose FIFL, a fault injecting approach to localizing faulty edits in evolving programs. Xuan & Monperrus [32] develop test case purification, which seperates test cases into small fractions to enhance the test oracle for fault localization. Le & Lo [13] consider the applicability of fault localization and develop an automatic method for predicting effectiveness with features in faulty programs.

Fault localization depends on a set of test cases. The cost of labeling and executing test cases is expensive. Gong et al. [9] design a diversity-maximization-speedup method for reducing the cost of test case labeling. Mei et al. [17] and Zhang et al. [34] focus on the test case prioritization, which ranks test cases to find faults in the early stage. Fault localization can be also used as a pre-phase in automatic source code repair to determine the faulty position in repairing methods, such as GenProg by Le Goues et al. [14] and Nopol by Demarco et al. [6].

[3]Software-artifact Infrastructure Repository, http://sir.unl.edu/

VIII. Conclusion

Spectrum-based fault localization aims to detect the exact position of faults in source code. In this paper, we propose MULTRIC, a learning-based approach to combining multiple ranking metrics. MULTRIC consists of two major phases, namely learning and ranking. In the learning phase, MULTRIC selects training pairs of faulty and non-faulty program entities to weight existing ranking metrics; in the ranking phase, MULTRIC combines the learned weights with existing ranking metrics to form the final ranking for a new faulty program. Experimental results show that MULTRIC can effectively localize Java faults, comparing with state-of-art ranking metrics.

In future work, we plan to further improve the effectiveness of our approach. We want to refine data selection strategy to build a more effective model with a smaller amount of training data. Moreover, we plan to empirically examine the performance and the generality of our work on other Java and C programs.

Acknowledgment

We thank Friedrich Steimann, Marcus Frenkel, and Rui Abreu for sharing their fault data.

References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[3] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96. ACM, 2005.

[4] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight bug localization with ample. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 99–104. ACM, 2005.

[5] V. Debroy and W. E. Wong. A consensus-based strategy to improve the quality of fault localization. *Software: Practice and Experience*, 43(8):989–1011, 2013.

[6] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39. ACM, 2014.

[7] M. Feng and R. Gupta. Learning universal probabilistic models for fault localization. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 81–88. ACM, 2010.

[8] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *The Journal of machine learning research*, 4:933–969, 2003.

[9] L. Gong, D. Lo, L. Jiang, and H. Zhang. Diversity maximization speedup for fault localization. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 30–39. IEEE, 2012.

[10] L. Gong, D. Lo, L. Jiang, and H. Zhang. Interactive fault localization leveraging simple user feedback. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 67–76. IEEE, 2012.

[11] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 167–178. ACM, 2008.

[12] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[13] T.-D. B. Le and D. Lo. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 310–319. IEEE, 2013.

[14] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[15] T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[16] D. Lo, L. Jiang, F. Thung, A. Budi, et al. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 2013.

[17] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *Software Engineering, IEEE Transactions on*, 38(6):1258–1275, 2012.

[18] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[19] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Database-aware fault localization for dynamic web applications. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 456–459. IEEE, 2013.

[20] S. J. Pan and Q. Yang. A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359, 2010.

[21] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 56–66. IEEE, 2009.

[22] R. E. Schapire, Y. Freund, P. Bartlett, W. S. Lee, et al. Boosting the margin: A new explanation for the effectiveness of voting methods. *The annals of statistics*, 26(5):1651–1686, 1998.

[23] R. E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine learning*, 37(3):297–336, 1999.

[24] F. Steimann and M. Frenkel. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 121–130. IEEE, 2012.

[25] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 314–324. ACM, 2013.

[26] M.-F. Tsai, T.-Y. Liu, T. Qin, H.-H. Chen, and W.-Y. Ma. Frank: a ranking method with fidelity loss. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 383–390. ACM, 2007.

[27] S. Wang, D. Lo, L. Jiang, H. C. Lau, et al. Search-based fault localization. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 556–559. IEEE Computer Society, 2011.

[28] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai. Effective fault localization using code coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference-Volume 01*, pages 449–456. IEEE Computer Society, 2007.

[29] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31, 2013.

[30] X. Xie, F.-C. Kuo, T. Chen, S. Yoo, and M. Harman. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In G. Ruhe and Y. Zhang, editors, *Search Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 224–238. Springer Berlin Heidelberg, 2013.

[31] J. Xu, Z. Zhang, W. Chan, T. Tse, and S. Li. A general noise-reduction framework for fault localization of java programs. *Information and Software Technology*, 55(5):880–896, 2013.

[32] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014.

[33] S. Yoo. Evolving human competitive spectra-based fault localisation techniques. In G. Fraser and J. T. de Souza, editors, *Proceedings of the 4th International Symposium on Search-Based Software Engineering*, September 2012.

[34] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 192–201. IEEE, 2013.

[35] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 765–784. ACM, 2013.