XAI4FL: Enhancing Spectrum-Based Fault Localization with Explainable Artificial Intelligence

Ratnadira Widyasari*, Gede Artha Azriadi Prana*, Stefanus A. Haryono*, Yuan Tian†, Hafil Noer Zachiary*, David Lo*
*School of Computing and Information Systems, Singapore Management University, Singapore
{ratnadiraw.2020,arthaprana.2016,stefanusah,hnzachiary.2019,davidlo}@smu.edu.sg
†School of Computing, Queen's University, Canada
y.tian@queensu.ca

ABSTRACT

Manually finding the program unit (e.g., class, method, or statement) responsible for a fault is tedious and time-consuming. To mitigate this problem, many fault localization techniques have been proposed. A popular family of such techniques is spectrum-based fault localization (SBFL), which takes program execution traces (spectra) of failed and passed test cases as input and applies a ranking formula to compute a suspiciousness score for each program unit. However, most existing SBFL techniques fail to consider two facts: 1) not all failed test cases contribute equally to a considered fault(s), and 2) program units collaboratively contribute to the failure/pass of each test case in different ways.

In this study, we propose a novel idea that first models the SBFL task as a classification problem of predicting whether a test case will fail or pass based on spectra information on program units. We subsequently apply eXplainable Artificial Intelligence (XAI) techniques to infer the local importance of each program unit to the prediction of each executed test case. Applying XAI to the failed test case, we retrieve information about which program statements within the test case that are considered the most important (i.e., have the biggest effect in making the test case failed). Such a design can automatically learn the unique contributions of failed test cases to the suspiciousness of a program unit by learning the different and collaborative contributions of program units to each test case's executed result. As far as we know, this is the first XAI-supported SBFL approach. We evaluate the new approach on the Defects4J benchmark dataset.

We compare the performance of our approach against five popular SBFL techniques: DStar, Tarantula, Barinel, Ochiai, and O^P. We measure their performance using the Top-K and *EXAM* scores. In particular, we focus on the result of the Top-1, which importance has been highlighted in automated program repair domain, where the proposed methods often assume perfect fault localization (i.e., the fault must be found at the first rank of the suspiciousness list). Our results show that our approach, named XAI4FL, has a statistically significant and substantially better performance in terms of Top-1 than the SBFL approaches. We also compare our approach with a simpler approach to get feature importance in a tree-based model (i.e., using the Mean Decrease in Impurity method). Our results show that XAI4FL statistically significantly outperforms the MDI method in Top-K and *EXAM* score. Our results and findings highlight that the utilization of XAI for fault localization can improve the overall results of fault localization techniques.

ICPC '22, May 16–17, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9298-3/22/05 …$15.00
https://doi.org/10.1145/3524610.3527902

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Fault Localization, Explainable Artificial Intelligence (XAI), Model-Agnostic Explanation Technique, Spectrum-based Fault Localization, Testing and Debugging

ACM Reference Format:
Ratnadira Widyasari*, Gede Artha Azriadi Prana*, Stefanus A. Haryono*, Yuan Tian†,, Hafil Noer Zachiary*, David Lo*. 2022. XAI4FL: Enhancing Spectrum-Based Fault Localization with Explainable Artificial Intelligence. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524610.3527902

1 INTRODUCTION

Debugging is a costly and time-consuming part of software development [43], yet it is an activity that developers typically have to do at various points of their software's life cycle. To mitigate the cost and time spent on debugging, researchers have proposed various automated fault localization techniques to help developers find faulty program units more quickly. Such techniques use information from failed and passed test cases, and output a list of locations in the program that are "suspicious", i.e., likely to cause the fault(s) being investigated. By doing so, automated fault localization techniques enable developers to quickly focus their attention on a specific part of the source code instead of checking the whole code base, increasing the developers' productivity [44].

One of the major families of fault localization techniques is spectrum-based fault localization (SBFL) [46]. This family comprises techniques such as Ochiai [1] and DStar [41], and is often used by automated program repair tools, such as PraPR [12] and CapGen [38]. Study by Pearson et al. [31] highlighted that among the SBFL techniques, DStar achieved the best performance, with the other four popular SBFL techniques (i.e., Ochiai, Barinel, Tarantula,

and O^P) being statistically indistinguishable from it. SBFL techniques process inputs in the form of program spectra, which are program traces containing details of passing and failing test cases, along with executed paths' information (e.g., executed statements). Following the intuition that a program unit covered by more failed tests but less passed tests is more likely to be faulty, SBFL applies a statistical formula to compute a suspiciousness score for each program unit based on the input program spectra [45]. The program units would be ranked from most to least suspicious, which can then be used by the developers to pinpoint the location of the fault.

Though SBFL techniques are well-researched in literature, they are not perfect yet [25]. One limitation of existing SBFL techniques is that they fail to fully capture the local relationship between the result of each executed test case and the program units involved. For instance, SBFL techniques, such as DStar [41], would assume that two failed test cases t_1 and t_2 contribute equally to the suspiciousness of one program unit executed in both test cases, regardless of the different number of program units involved in two test cases. While in practice, one failed test case may be more valuable than others as it only touches a few program units. Secondly, the fact that program units may collaboratively contribute to the result of each test case in different ways is ignored in many existing SBFL techniques. To illustrate, consider a test suite that includes 3 test cases, t1, t2, and t3, where t1 is a failed test case, and t2 and t3 are passing test cases. Test case t_1 executes ten program units but fails due to the buggy code in two connected program units p_1 and p_2. Meanwhile, consider the passed test cases t_2 and t_3 which run program units p_1 and p_3 for t_2, and p_2 and p_3 for t_3 respectively. Test cases t_2 and t_3 passed successfully as they do not contain both p_1 and p_2 in a single test case. This example highlights that p_1 and p_2 collaboratively cause the failure of the test case. Such cases are not considered by many existing SBFL techniques.

To overcome the aforementioned limitations, we propose the usage of eXplainable Artificial Intelligence (XAI) techniques to learn the local relationship between program units and the failure/pass outcome of each test case. XAI aims to gain insights into the black-box machine learning models and explain their output, e.g., explaining how input feature values lead to the predicted label of a specific data point and identifying important features [4]. Applications of XAI in software engineering are still minimal, with only a few studies have explored their potential usage in explaining defect prediction models [15, 19, 36]. Defect prediction aims to find the potential buggy files, commits, etc. based on historical data of prior bugs. Meanwhile, fault localization aims to find the root cause location (typically buggy statements) of failures based on available test data. The two lines of work consider different inputs and are deployed in different phases of software development. Due to these differences, they are studied as separate lines of work.

XAI can be applied to improve the performance of software engineering tasks due to several reasons [4]. For example, XAI can be utilized to provide justification. In this approach, XAI is used to ensure that Artificial Intelligence based decisions are not made erroneously. Meanwhile, other than using XAI for an explanation, XAI can also be used to enhance the predictive performance of the existing model. For example, previous work by Wattanakriengkrai et al. [36] utilized XAI to get better predictive performance for defect prediction. In our study, we do not use XAI for their explanation capability. Rather, our goal of XAI utilization is to achieve better results and accuracy for fault localization tasks.

Our intuition in utilizing XAI for fault localization is that XAI techniques can learn the local importance of program unit(s) to each test result by modeling the SBFL problem as a classification task, where each data point is an executed test case, data point's label is the result of the test, and input features are the corresponding program spectra. Such design allows us to learn how program units collaboratively contribute to each test result addressing the second limitation of many SBFL methods (i.e., not taking into account that program units can collaboratively contribute to the failure/pass test case). For instance, given a model trained to predict test results, and a failed test case t_1. XAI techniques may assign higher importance scores to program unit (feature) p_1 and p_2 based on the patterns captured by the model and their execution status (feature value), e.g., both program units are executed in test t_1, indicating that p_1 and p_2 collaboratively contribute to the prediction of t_1. In this design, since each pair of program units and the failed test case will be assigned a different importance score, the first limitation of SBFL (i.e., all failed test cases have equal contribution) is also resolved as not all failed tests are equally important for calculating the suspiciousness of each program unit.

Our XAI-supported SBFL approach (for faulty statements localization), i.e., XAI4FL, contains three main steps:

(1) We run a machine learning algorithm with the goal of classifying whether a test case is failed or passed.
(2) We use an XAI technique to identify important statements score which acts as an important factor in the prediction of the trained model for each failed test case.
(3) We calculate the suspiciousness score of each candidate statement by combining its importance scores from all failed test cases. These scores are then used to rank the faulty statements from the most to the least suspicious.

In this study, our goal is to design, implement, and investigate the effectiveness of XAI4FL for fault localization. Specifically, we aim to answer the following research questions:

(1) RQ-1: **How effective are the variants of XAI4FL?** We seek to investigate the effectiveness of XAI4FL in localizing fault. We inspect three variants of our XAI4FL to identify the variant that works the best on localizing fault.
(2) **RQ-2: How does XAI4FL improve the fault localization results compared to a simpler approach in getting the feature importance?** We compare the performance of the model-agnostic explanation technique (i.e., XAI4FL) with a simpler approach (i.e., by using Mean Decrease in Impurity (MDI) for tree-based model [29]) to get feature importance. We aim to investigate whether our XAI-based method can provide better results compared to a simpler approach.
(3) **RQ-3: How does XAI4FL compare with traditional SBFL techniques in terms of effectiveness?** We compare XAI4FL performance against popular SBFL techniques. This comparison is aimed to check whether XAI4FL achieves better results in localizing faults compared to SBFL techniques.

We evaluate XAI4FL to identify faulty statements on the Defects4J V1.2.0 dataset [18]. Defects4J contains 395 faulty versions from six Java projects and has been used widely as a benchmark

for fault localization and automated program repair [31]. We evaluate the fault localization performance by measuring the EXAM score [40], which is popularly used in many previous studies [2, 23, 31, 42]. Motivated by findings of Parnin and Orso study [30] where the absolute rank (i.e., Top-K score) is more important than the percentage ranking (i.e., *EXAM* score), we also calculate the Top-K scores, which is also used in many previous studies [23, 31]. For the Top-K scores, the value of K must be set to a specific value. One of the value that has been used in many previous studies [20, 23] is k=1 (i.e., Top-1 score). The Top-1 accuracy is crucial as many works in automated program repair [9, 14, 27] assume perfect fault localization (i.e., the fault is positioned in the top-1 prediction). Due to this assumption, automated program repair may not work well if the fault is not found in the Top-1 result. Our experiment results show that XAI4FL statistically significantly outperforms five popular SBFL techniques [42] (i.e., Tarantula [16], Ochiai [1], O^P [28], BARINEL [2], and DStar [41]) in terms of Top-1 accuracy.

In summary, our contributions are as follows:

(1) We propose a new fault localization technique based on the combination of machine learning and model-agnostic explanation technique. We are the first to promote the usage of XAI (eXplainable Artificial Intelligence) for fault localization.
(2) We conduct an evaluation of our proposed approach on a dataset that consists of 395 faults from 6 projects and show that our approach statistically significantly and substantially outperforms popular SBFL techniques on Top-1.
(3) We highlight our findings from XAI4FL, which include its current limitation and challenges that should be addressed in future work.

The rest of this paper is organized as follows. In Section 2, we briefly explain SBFL techniques and model-agnostic explanation techniques. Section 3 provides details of our proposed approach, while the experiment settings of our study is presented in Section 4. We show the experiment results and analyze them in Section 5. In Section 6, we discuss the possible limitations of our approach and how to mitigate them. We discuss the threats to validity in Section 7. In Section 8, we summarize the related works. Finally, we conclude our work and present future directions in Section 9.

2 BACKGROUND

2.1 Spectrum-based Fault Localization Techniques

Spectrum-based fault localization (SBFL) technique is a popular and actively researched family of fault localization techniques [42]. SBFL uses a statistical formula to measure the suspiciousness of each program unit (e.g., statement, function) being the cause of the investigated fault(s), based on test results and the corresponding execution traces. An execution trace is a sequence of program units invoked while executing a test case. Program units are then ranked based on the calculated suspiciousness score from highest to lowest.

Many SBFL techniques have been proposed [1, 41]. Although they use distinct formulas to calculate the suspiciousness score of each program unit, the underlying assumption is the same, i.e., the program units executed in more failed test cases and fewer passed test cases are more likely to be faulty. The five most popular and well-studied SBFL techniques are Tarantula [16], Ochiai [1], O^P [28], Barinel [2], and DStar [41]. A previous study by Pearson et al. [31] focused on evaluating the effectiveness of various SBFL techniques. They reported that among the 5 popular SBFL techniques mentioned earlier, DStar, Tarantula, Barinel, and Ochiai produces approximately equivalent performance on the Defects4J data.

Using these popular SBFL techniques, we calculate the suspiciousness score of each program unit s using the following formula:

$$Ochiai(s) = \frac{n_f(s)}{\sqrt{n_f.(n_p(s)+n_f(s))}} \qquad Tarantula(s) = \frac{\frac{n_f(s)}{n_f}}{\frac{n_f(s)}{n_f}+\frac{n_p(s)}{n_p}}$$

$$DStar(s) = \frac{n_f(s)^2}{n_p(s)+(n_f-n_f(s))} \qquad Barinel(s) = 1 - \frac{n_p(s)}{n_p(s)+n_f(s)}$$

$$O^P(s) = n_f(s) - \frac{n_p(s)}{n_p+1}$$

where n_f denotes the number of total failing test cases, n_p denotes the number of total passing test cases, $n_f(s)$ denotes the number of failing test cases that execute program unit s, and $n_p(s)$ denotes the number of passing test cases that execute program unit s.

2.2 XAI and Model-Agnostic Explanation Techniques

XAI is a rising research field of interest, aiming to provide explanations for machine learning models. Recent studies have shown the importance of having such explanations in maintaining transparency, trust, and fairness in the Machine Learning decision-making process, as well as in facilitating better model design [11].

XAI techniques can be model-agnostic or model-specific. Model-agnostic explanation techniques work for any machine learning model, while model-specific techniques rely on a certain model structure. Model-agnostic explanation techniques can be local, aiming to understand why a trained model made a certain prediction on a specific instance, or global, aiming to understand the model's behavior as a whole. In this study, we utilize local model-agnostic explanation techniques to identify the local contribution of each feature in a given input to its prediction.

Several model-agnostic explanation techniques have been proposed, with the two most popular local model-agnostic are Local Interpretable Model-Agnostic Explanations (LIME) [32] and SHapley Additive exPlanations (SHAP) [26]. LIME implements a local surrogate model to explain individual prediction, with the surrogate model being an interpretable model trained to approximate the predictions of the underlying black-box model. LIME main idea is that explanation may be derived locally from records generated randomly in the neighborhood of the instance that has to be explained. On the other hand, SHAP is based on coalitional game theory and measures the feature importance while also considering interaction within features. There are currently several strategies to compute SHAP values, such as KernelSHAP, which can produce local explanation from any models, and TreeSHAP which can compute SHAP values from tree ensemble models such as Random Forest and Decision Tree, and Gradient Boosted Tree in polynomial-time.

In software engineering, we find that XAI techniques are most commonly applied to improve the interpretability of a machine-learning based automation approach. Specifically, XAI is often utilized to improve defect prediction [15, 19, 36]. Different from existing approaches, our idea is to utilize model-agnostic explanation

techniques to improve the effectiveness of SBFL. The aim of fault localization is to find the root cause location (e.g., buggy statements) of failures based on available test data. Meanwhile, defect prediction aims to find the files, commits, etc. that are prone to a bug, based on historical data of prior bugs. Defect prediction and fault localization are studied as separate lines of work as they need different inputs and are deployed on different phases of software development.

3 PROPOSED APPROACH

Our main idea is to enhance SBFL by incorporating a machine learning model and model-agnostic explanation technique to provide a more accurate ranking of a program unit from being a root cause of a set of failures. Following many previous studies [5, 31, 48], the granularity of the program unit that we consider in this study is a statement. As a note, our method can also be applied for method-level localization by changing the data input. The summarized workflow of our approach is as follows. Given inputs in the form of program spectra, we train a classification model that predicts whether each input is from a passed or failed test case. Then, we apply model-agnostic explanation techniques to infer the local importance score of each statement to predict whether or not a test will fail. Finally, we combine the importance scores of the features from failed test cases to infer the suspiciousness of each statement. Figure 1 presents an overview of our approach. Details on each component are provided in the rest of this section.

(Step 1) Obtain program spectra from test cases. In this step, we take as input a faulty program, and a set of test cases. We use GZoltar[1] to run the test cases and collect their execution traces. GZoltar outputs two files: (1) a matrix file that contains the execution traces (i.e., series of program units that are executed by the passed and failed test cases), and (2) a spectra file that contains a list of all statements that executed by test cases. Then, we combine these two outputs (i.e., the matrix and spectra files) for easier processing by the classification algorithm. An example of the combined result between the matrix and spectra files is shown in Figure 1.

Each row in this combined program spectra represents a single test case. The failed test case is labeled as negative ("−") and the passed test case is labeled as positive ("+"). Meanwhile, the columns located in between the id and label columns represent the statements (e.g., column named *Node#375, SortOrder#74*). The value for these statements on each row is either 1 or 0, where 1 indicates a program unit executed by the test case, and 0 otherwise. This combined program spectra representation is then used as the input for the training process of the machine learning classifier.

(Step 2) Train the machine learning classifier. Given a program spectra (logs of program units executed by a set of failing and passed test cases), we train a model to approximate a function to measure the likelihood of whether the execution of a program unit determines the outcome of the test case. This function is then used to identify the important program units that are likely to be responsible for the failing test case. These program units are likely to be the locations where the faults reside. For this purpose, we choose Random Forest for our classifier, as it has been reported to outperform other basic classifiers in many Software Engineering tasks [15, 19, 36]. Moreover, we have also tried other classifiers such as Support Vector Machine, and they perform worse than Random Forest in our dataset. The machine learning classifier is then used to determine whether a given input test case is a passed or failed test case.

[1]http://www.gzoltar.com/

Table 1: Example of program spectra.

id	SortOrder#74	SortOrder#75	...	Node#375	Label
0	1	1	...	0	+
1	0	0	...	0	+
2	1	1	...	1	-

(Step 3) Apply model-agnostic explanation technique. We apply a local model-agnostic explanation technique to identify which features (statements) are important in each failed test case. In other words, we want to get the program statements that have the biggest effect in making the machine learning model predict a test case as failed. We consider two model-agnostic technique, which are SHAP and LIME (c.f., Section 2). After an initial test, we pick SHAP as it provides better performance on our dataset. Specifically, we use TreeSHAP, an algorithm to compute SHAP values for tree ensemble models such as Random Forest. Consider a failed test case t_1, which contains statements s_1 to s_{100}. Applying TreeSHAP to this test case, it calculates the importance score of each statement. Figure 2 shows an example of Top-5 most important statements for a sample failed test case returned by TreeSHAP. In this example, s_{52} is the most important statement to predict that the sample test case is failed, as indicated by s_{52} having the highest importance score calculated by TreeSHAP.

(Step 4) Aggregate scores and rank the results. The model-agnostic explanation technique outputs the importance score of statements from all the failed test cases. In this step, we combine these importance scores by considering three variants:

- **Mean variant**: for each statement, calculate the average of its importance scores on all failed test cases.
- **Max variant**: for each statement, calculate the maximum of its importance scores on all failed test cases.
- **Min variant**: for each statement, calculate the minimum of its importance scores on all failed test cases.

Finally, we rank the statements based on the combined scores from the highest to the lowest score. This ranking can then be used as the reference for developers in fixing the faults or for the automated program repair to get the location of the fault.

4 EXPERIMENT SETTINGS

4.1 Dataset

We evaluate our approach on version 1.2.0 of the Defects4J dataset [18] that consists of 395 faulty versions of six Java projects (i.e., JFreeChart, Google Closure compiler, Apache Commons Lang, Apache Commons Math, Mockito, and Joda-Time). We choose Defects4J because it contains real faults and has been widely used as a benchmark to measure the performance of fault localization techniques and automated program repair techniques [21, 31]. We note that several other datasets have been used to evaluate fault localization techniques in previous studies [17, 24, 41]. However, these other datasets are mostly comprised of artificial faults (e.g., [24] dataset

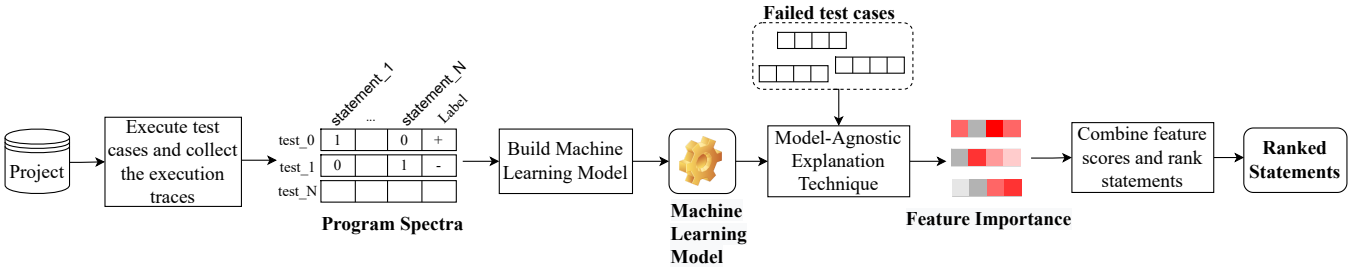

Figure 1: Overview of our approach

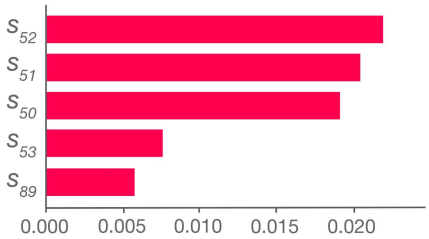

Figure 2: Example of TreeSHAP output showing the Top-5 most important statements for the prediction on a sample failed test case.

Table 2: Statistics of Defects4J v1.2.0

Projects	KLOC	# Faults	# Tests
JFreeChart	96	26	2,205
Google Closure compiler	90	133	7,927
Apache Commons Lang	22	65	2,245
Apache Commons Math	85	106	3,602
Mockito	45	38	2,115
Joda-Time	28	27	4,130
Total	**366**	**395**	**22,224**

consists of only 35 real faults while the other 165 are artificial faults). Considering that the evaluation of fault localization in artificial faults may not be a useful predictor for fault localization in real faults [31], we decide to utilize the real faults in the Defects4J dataset. Table 2 shows the basic statistics of our evaluation dataset.

4.2 Evaluation Metric and Experimental Setting

We consider two evaluation metrics for measuring the effectiveness of fault localization techniques, i.e., Top-K score and EXAM score. These metrics are widely used in literatures [17, 20, 24, 30, 31, 41]. In particular, we focus our evaluation in the Top-1 result of the fault localization techniques. This is due to the Top-1 prediction being crucial for automated program repair which often assumes that the fault is positioned in the Top-1 prediction [9, 14, 27].

Top-K score first counts the number of successfully localized faulty versions of projects by checking the Top-N ranked results, and then divides it by the total number of faulty versions. We use this metric as a previous study by Parnin and Orso [30] found that absolute ranking is more important than percentage ranking for fault localization. Their evaluation reveals that if the correct faulty program unit is not shown in the top-ranked list, developers would not find the fault localization result useful. In real projects, such as those in Defects4J, multi-statement faults exist. We treat a localization of a faulty version (ranking of suspicious statements) as successful if any faulty statement(s) that need to be repaired are found in Top-K most suspicious statements. For statements with the same suspiciousness score, we calculate an average rank using $(\frac{n}{2}) + (k-1)$, where n is the number of statements that share the same suspiciousness score and k is the best rank of the statement [31]. As an illustration, consider the statements s_1, s_2, and s_3 with a suspiciousness score of 1, which is the highest among all statements. The statements s_1, s_2, s_3 have an average rank of 1.5 (i.e., $(\frac{3}{2}) + (1-1) = 1.5$). Other than measuring the Top-1 prediction, we also measure the Top-5 and Top-10 predictions.

The *EXAM* score is the percentage of program units that needs to be checked before we reach the first faulty program unit. The formula for calculating the *EXAM* score is as follows:

$$EXAM\ Score = \frac{Rank\ of\ the\ first\ faulty\ program\ units}{Total\ number\ of\ program\ units}$$

Lower *EXAM* score indicates a better performance of the fault localization technique. The *EXAM* score ranges from 0 to 1 inclusive.

As mentioned in Section 3, we pick Random Forest [7] as the machine learning algorithm and SHAP [26] as the model-agnostic explanation technique. To train the machine learning classifier, we first split the input data into train and test sets. Specifically, we perform stratified sampling strategy [3] to randomly pick 80% input program spectra as the train set and the remaining 20% as the test set, while making sure that the train and test dataset contain the same ratio of failed test cases. We believe that such balanced data would help to train a model that can better generalize into the whole dataset. We use this combination as it performs the best in our experiment.

For each fault, we built a single machine learning model using the corresponding program spectra (i.e., sequences of statements that were executed by the test cases) as inputs. As we consider 395 faults in our evaluation, there are 395 separate models. The average from the precision, accuracy, and F1 for the models test results are 71%, 96%, and 73% respectively. As a baseline, we utilize five popular SBFL techniques, namely DStar, Barinel, Ochiai, Tarantula, and O^P. To collect the program spectra and implement these baselines techniques, we use the code provided by Pearson et al. [31]. The code of our implementation is publicly available.

4.3 Research Questions

4.3.1 RQ-1. : How effective are the variants of XAI4FL?

In RQ-1, we inspect the three variants of XAI4FL: mean, max, min (see Section 3). These three variants are based on the approach on how we aggregate the results from the feature importance scores produced by the model-agnostic technique. We run the three variants of XAI4FL using Defects4J data. To evaluate their performances, we use *EXAM* and Top-K score metrics. After retrieving these performance metrics, we utilize Wilcoxon rank-sum test [39] following previous work by Le et al. [23]. Wilcoxon rank-sum test is used to identify whether the differences in performance between the three variants are statistically significant. We use 5% significance level, meaning that if the p-value is lower than 0.05, we can reject the null hypothesis and conclude that there is a statistically significant difference between the variants. This statistical test is applied for both Top-k and *EXAM* score metrics for each variant of XAI4FL.

Furthermore, we also compute Cliff's d effect size [10], following the work by Le et al. [10]. Cliff's d effect size is commonly used to measure how substantially different the two techniques are. Using this, we measure whether the differences between the three XAI4FL variants are substantial. To interpret the effect size, we use the following interpretation: *negligible* if d < 0.147; *small* if 0.147 ≤ d < 0.33, *medium* if 0.33 ≤ d < 0.474, and *large* if d>=0.474 [13].

4.3.2 RQ-2. : How does XAI4FL improve the fault localization results compared to a simpler approach in getting the feature importance?

For the second research question, our goal is to confirm whether our proposed XAI4FL performs better than a simpler approach to get feature importance. In RQ-1, we use the model-agnostic explanation technique, specifically SHAP, to get the feature importance score. Meanwhile, in our RQ-2 investigation, we use a simpler approach to get feature importance. For this purpose, we utilize the already built nodes from the tree-based model. To get the feature importance score, we add up the weighted impurity decreases for all nodes and average them over all trees. This method to get the feature importance is called Mean Decrease in Impurity (MDI) [29]. After we get the feature importance score using the MDI method, we rank the statements based on the score from highest to lowest. The feature importance score produced using this method is not based on the individual test case. Rather, it is based on the whole machine learning model. In other words, we only get one score for every feature/statement. Thus, we do not need to aggregate the feature importance score like in XAI4FL. Then, we compare the results from XAI4FL with the MDI method.

To compare the two approaches, we use *EXAM* and Top-K score as the performance metrics. To ensure a fair comparison, we use the same random forest models that are used for the model-agnostic technique in XAI4FL (RQ-1). Then, we determine whether the differences between the two techniques are significant by using the Wilcoxon test [39] and whether the differences are substantial by calculating Cliff's d effect size [10]. The Wilcoxon test and Cliff's d effect size interpretation that we use for this RQ-2 is the same as the one that we use for RQ-1 (c.f. Section 4.3.1).

4.3.3 RQ-3. : How does XAI4FL compare with traditional SBFL techniques in terms of effectiveness?

In RQ-3 we investigate how XAI4FL performs compared to the SBFL techniques. We consider five popular SBFL techniques for this comparison, which are DStar, Barinel, Ochiai, Tarantula, and O^P. These five techniques are well studied and commonly used in prior studies on SBFL [31, 37, 47]. In particular, a study by Pearson et al. [31] highlighted that DStar performs the best with the other four techniques having comparable performance. We run these five SBFL techniques using the same Defects4J dataset that we use in prior RQs. We also utilize two performance metrics, which are *EXAM* and Top-K score. After obtaining the performance metrics from the SBFL techniques, we then compare their results with XAI4FL. We use the same statistical test and effect size test as RQ-1.

Table 3: Top-K scores of XAI4FL techniques.

Technique	Top-1	Top-5	Top-10
XAI4FL (Max variant)	13.5% (53)	31% (124)	42% (166)
XAI4FL (Mean variant)	13% (50)	32% (126)	42% (166)
XAI4FL (Min variant)	13% (51)	31% (123)	41% (163)

Table 4: *EXAM* scores of XAI4FL, MDI, and SBFL techniques. Results in bold shows the best performance.

Technique	*EXAM* Score
XAI4FL (Max variant)	**0.031873**
XAI4FL (Mean variant)	0.031995
XAI4FL (Min variant)	0.032205
MDI	0.073372
DStar SBFL	0.036680
Barinel SBFL	0.037718
Ochiai SBFL	0.036815
Tarantula SBFL	0.037718
O^P SBFL	0.044025

5 EVALUATION RESULTS

5.1 RQ-1 How effective are the variants of XAI4FL?

We evaluate the three variants of our approach by utilizing three feature importance scores aggregation methods: Max, Min, and Mean variant. Table 3 and Table 4 show the Top-K (i.e., Top-1, 5, and 10) scores and *EXAM* scores of the three variants, respectively. Overall, the Max variant achieves the best performance when considering the Top-1, Top-10, and *EXAM* score. The difference in the number of faults localized at Top-K within three aggregation methods span from 0 to 3. The highest number of real faults (from 395 faults in Defects4J) that can be localized at Top-1 is 53 (Max variant).

We pair variants of XAI4FL against each other (e.g., *mean* vs. *max*, *mean* vs. *min*, and *max* vs. *min*). Then we run statistical test for each pair to get whether the differences between them are statistically significant. Based on the statistical test, we find that the differences between the *EXAM* score of those three variants are not statistically significant for all the pairs, as shown by the p-value of statistical test results that are higher than 0.05. The effect sizes are also negligible which are indicated by the effect size that is less than 0.147. For the Top-K results, there are also no statistical and substantial differences between the variants.

All of the 50 faults that are localized in the Top-1 by the *mean variant* variant are also localized in the Top-1 for the *max variant*

and *min variant* variants. Meanwhile, the improvement of Top-1 in *max variant* and *min variant* variants from the *mean variant* do not overlap. There are three additional faults (i.e., total of 53 faults) that are localized in Top-1 by the *max variant* variant. Meanwhile, for the *min variant* variant, there is one additional fault (i.e., total of 51 faults) that is localized in the Top-1.

The EXAM score and the Top-1 results from the mean variant have lower performance than the max variant. This may happen because, in the mean variant, the contribution of each test case will be equal as they are averaged. Meanwhile, in the max variant, the results will depend on the test cases that have higher feature importance scores. For the remainder of the RQs (RQ-2 and RQ-3), we use the *max variant* of XAI4FL variant, which is the best performing XAI4FL, for our comparisons.

RQ-1 Findings: The XAI4FL variant that produces the best results to localize faulty statement is *max variant*. It achieves the best Top-1, Top-10, and *EXAM* score. However, the differences are not statistically significant with the other two variants (i.e., the *mean variant* and *min variant*).

5.2 RQ-2 How does XAI4FL improve the fault localization results compared to a simpler approach in getting the feature importance?

In this RQ-2 we compare XAI4FL with a simpler approach to get the feature importance score. The simpler approach that we choose for this task is the MDI (Mean Decrease in Impurity) method [29] that is obtained from the Random Forest machine learning model. The Top-K results from this method are shown in the second row of Table 5 (i.e., row "MDI"). For all the K values (i.e., 1, 5, and 10), MDI gives lower performance results compared to the XAI4FL techniques. The number of faults that are localized in Top-1 is 43, while in XAI4FL, the number of localized faults for Top-1 is 53 (23.26% improvements). For the Top-5 and Top-10, MDI localizes 113 and 148 faults respectively, compared to the XAI4FL that can localize 124 and 166 faults for Top-5 (9.73% improvements) and Top-10 (12.6% improvements) respectively. The improvement in Top-K by using XAI4FL compared to the MDI to get feature importance ranges from 9 to 18 faults, with the biggest improvement shown in Top-10 (18 fault improvements in *max variant* XAI4FL).

Table 5: Top-K scores of XAI4FL and MDI techniques.

Technique	Top-1	Top-5	Top-10
XAI4FL (Max variant)	13.5% (53)	31% (124)	42% (166)
MDI	11% (43)	29% (113)	37.5% (148)

The results of statistical test and effect size test between XAI4FL and MDI approach can be found in Table 6. Based on the statistical test, we observe that the differences between XAI4FL and MDI are statistically significant. It shows that XAI4FL statistically significantly outperforms the performance of the simple approach on Top-1, Top-5, and Top-10.

Figure 3 represents the faults that are successfully localized in Top-1 by XAI4FL and by the simpler approach (i.e., MDI, the feature importance score for tree model). From the 43 faults that are localized in Top-1 by the MDI approach, 27 (63%) of them are already localized in Top-1 by XAI4FL. Analyzing the other 16 faults that are localized in Top-1 using MDI but not on Top-1 in XAI4FL, we find that 69% of them are localized in the Top-5 by XAI4FL. Moreover, we also observe that there are 26 faults that are localized in Top-1 by XAI4FL but not by MDI.

Table 6: Cliff's d effect sized and Wilcoxon test results: XAI4FL VS. MDI (simpler approach than model-agnostic technique to get feature importance). "*" indicates that the differences between the distribution of Top-K scores is statistically significant (at significance level of 5%). "(Neglible(N)/Small(S)/Medium(M))" denotes the categorization of effect size.

Technique	MDI		
	Top-1	Top-5	Top-10
XAI4FL (Max variant)	0.16* (S)	0.10* (N)	0.09* (N)

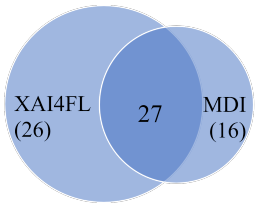

Figure 3: Venn diagram of Top-1 results for XAI4FL vs. simpler approach to get feature importance (MDI)

The *EXAM* score result for the MDI approach is shown in Table 4. Comparing the results of the XAI4FL and MDI approach, XAI4FL has a lower *EXAM* score which indicates a better performance. The best value of *EXAM* score from XAI4FL is 0.0319, while the MDI only achieves a score of 0.0734 which is more than twice the EXAM score value of XAI4FL. The p-value from the statistical test comparing the *EXAM* score of the XAI4FL *max variant* with MDI is lower than 0.05 (p-value = 0.015) which highlights that the differences between XAI4FL and MDI are statistically significant.

The overall results show that XAI4FL outperforms MDI. One possibility of the lower performance of the MDI is due to the feature importance scores of the simpler model are dependant on the model. This results in the feature importance scores being based on the whole training data rather than based on only the failed test cases. This is different than in XAI4FL where the feature importance scores are based on the test cases that are failed (i.e., specific per instance). These results highlight that simpler models which are derived from the machine learning model are not specific enough to explain the individual failed instances. Another reason why the MDI method gives lower performance is because of the limitation of MDI itself, as it is biased toward correlated features [35]. In the event of correlated features, it might choose one feature while ignoring the relevance of the other, which can lead to wrong conclusions.

RQ-2 Findings: XAI4FL statistically significantly outperform MDI method (i.e., simpler approach to get feature importance in tree-based model) in both Top-K and *EXAM* scores. The lower results of MDI method may be due to the feature importance scores in MDI which represent the whole model rather than only the failed instances.

5.3 RQ-3 How does XAI4FL compare with traditional SBFL techniques in terms of effectiveness?

We compare XAI4FL with five well-known SBFL techniques: DStar, Barinel, Ochiai, Tarantula, and O^P. Table 7 shows the performance comparison in terms of Top-K score between the SBFL techniques and XAI4FL. We can observe that the number of real faults localized at Top-1 by XAI4FL is approximately twice the number of localized faults by the SBFL techniques. The number of faults that are localized in Top-1 using SBFL techniques ranges from 24 to 27, while the number of faults localized by XAI4FL ranges from 50 to 53. This highlights that XAI4FL can provide 23 to 29 localized fault improvements in the Top-1. We believe that such improvement at the Top-1 score is important as many existing automated program repair approaches assume perfect fault localization, where the fault can be localized at Top-1 [9, 14, 27].

Table 7: Top-K scores of XAI4FL and SBFL techniques.

Technique	Top-1	Top-5	Top-10
XAI4FL (Max variant)	13.5% (53)	31% (124)	42% (166)
DStar SBFL	7% (27)	32% (127)	42% (166)
Barinel SBFL	6% (24)	31% (123)	42% (166)
Ochiai SBFL	7% (27)	32% (128)	42% (168)
Tarantula SBFL	6% (24)	31% (123)	42% (166)
O^P SBFL	7% (26)	30% (119)	40% (157)

Table 8: Cliff's d effect sized and Wilcoxon rank-sum test results: XAI4FL VS. DStar (a traditional SBFL Approach).

Technique	DStar		
	Top-1	Top-5	Top-10
XAI4FL (Max variant)	0.39* (M)	0.02 (N)	0.006 (N)

We also conduct statistical test for the SBFL technique to compare the result with XAI4FL. Specifically, we compare XAI4FL against the SBFL technique that achieves the best Top-1 and EXAM score, DStar. The results of the Wilcoxon rank-sum test and Cliff's d effect size are shown in Table 8. We find that XAI4FL outperforms DStar at Top-1 with a statistically significant difference. Moreover, the effect sizes indicate that the differences between the Top-1 scores of DStar and XAI4FL are substantial (i.e., medium effect size). The results of our approach at Top-5 and Top-10 are on par with the results from DStar, with no statistically significant difference. The results for DStar are also inline with four other popular SBFL technique where XAI4FL statistically and substantially improve the results in Top-1 and achieve comparable results in Top-5 and Top-10.

Figure 4 represent the faults that are localized in top-1 for XAI4FL and DStar. We choose DStar as a representative as it achieves the best result among the SBFL techniques. Based on the diagram, we observe that the majority of faults that are localized in the Top-1 of DStar are also localized by XAI4FL. From 27 faults that are successfully localized in Top-1 by DStar, there are 16 faults (60%) that are localized in Top-1 by XAI4FL. Meanwhile, for the other 11 faults that are localized on Top-1 only by DStar, 10 out of the 11 faults are localized within Top-5 predictions by XAI4FL (another 1 fault is localized in Top-9). This finding indicates that for cases that are perfectly localized in DStar, XAI4FL can also provide good results. On the contrary, 19% of the faults that are localized in Top-1 by XAI4FL but not by DStar are localized by DStar with a rank higher than 5. This indicates that there are cases where XAI4FL can localize but DStar cannot. Moreover, in the 37 faults localized in Top-1 by XAI4FL but are not localized in Top-1 by DStar, there are many features that share the same high suspiciousness score in DStar, resulting in average the ranking of the statements that share suspiciousness score (e.g., ranked in 1.667). This makes the localization in Top-1 not possible in some of DStar cases.

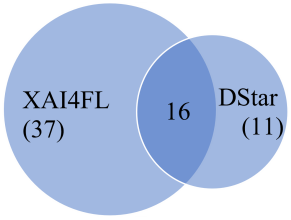

Figure 4: Venn diagram of Top-1 results for XAI4FL vs. DStar

Table 9 shows the detailed Top-K scores achieved by our approach and DStar on each project. For most projects (4 out of 6), we observe big improvements in terms of Top-1 score in our approach compared to DStar to our approach, i.e., the number of faults that are localized at Top-1 using our model is more than twice the number of that localized by DStar. For project *Lang* and *Time*, the number of successfully localized faulty versions at Top-1 increases 4 to 5 times when using our approach compared to DStar.

Table 9: Top-K results for detailed project of XAI4FL (max variant) and DSTAR (a traditional SBFL approach). Results in bold is the best for each project.

Project	Technique	Top-1	Top-5	Top-10
Chart	XAI4FL	**7.7% (2)**	**38.5% (10)**	**53.8% (14)**
	DStar	3.8% (1)	26.9% (7)	50.0% (13)
Closure	XAI4FL	**6.0% (8)**	15.0% (20)	24.1% (32)
	DStar	5.3% (7)	**18.0% (24)**	**25.6% (34)**
Lang	XAI4FL	**18.5% (12)**	44.6% (29)	58.5% (38)
	DStar	3.1% (2)	**46.2% (30)**	**61.5% (40)**
Math	XAI4FL	**18.9% (20)**	**39.6% (42)**	**48.1% (51)**
	DStar	10.4% (11)	34.9% (37)	45.3% (48)
Mockito	XAI4FL	13.2% (5)	28.9% (11)	42.1% (16)
	DStar	13.2% (5)	**39.5% (15)**	42.1% (16)
Time	XAI4FL	**22.2% (6)**	44.4% (12)	55.6% (15)
	DStar	3.7% (1)	**51.9% (14)**	55.6% (15)

Upon further analysis, we find that projects *Lang* and *Math* (i.e., projects that have the biggest number of faults localized in Top-1 using XAI4FL) have the lowest number of average failed test cases,

which are 1.9 and 1.6 respectively. Meanwhile, *Lang* and *Math* projects also have the lowest average number of a passed test case. The ratio between the failed test cases and the number of test cases for these two projects is also the highest compared to other projects. These number of failed and passed test cases, and the ratio between the failed and number of test cases, may be one of the reasons on why projects *Lang* and *Math* achieve good results using XAI4FL. As the ratio of failed test cases and passed test cases on these two projects are higher than the others (i.e., there are fewer differences in the number of failed and passed test cases), the data for these projects are more balanced, resulting in better performance.

Meanwhile, *Closure* project has the lowest performance compared to the other projects. We observe that this project has the highest number of features (i.e., the number of statements that are executed by test cases). *Closure* project has an average number of features of 16,538, compared to other projects which number of features range from 850 to 5,150. Having many features will increase the dimension of search space for the problem. This may cause the model to suffer from Curse of Dimensionality [8] and overfitting problem. We discuss how we can mitigate this problem in Section 6.

Table 4 shows the performance of our approach and SBFL techniques in terms of *EXAM* score. The best result between the SBFL techniques is achieved by DStar, while O^P gives the lowest performance. We observe that all the three variants of XAI4FL achieve a slightly lower *EXAM* score than all the SBFL techniques, indicating better performance. However, the differences are not significant.

RQ-3 Findings: XAI4FL gives better results in both Top-1 and *EXAM* scores compared to the traditional SBFL techniques. The improvements for the Top-1 results are statistically significant and substantial. Analyzing the improvements in specific projects, we observe that XAI4FL can localize two to five times the number of faults in Top-1 compared to using traditional SBFL techniques.

6 DISCUSSION

In this section, we discuss the limitation of our approach and how they can potentially be addressed in the future.

Failed test cases may have different contributions. RQ-2 and RQ-1 results imply that not all failed instances contribute equally. As an example, consider the mean variant of XAI4FL (RQ-1) results where all the feature importance scores are divided equally. This mean variant shows lower performance results compared to the max variant of XAI4FL which only takes the maximum feature importance score. Furthermore, consider the results of the simpler approach to get a feature importance score (RQ-2) where the scores are not based on individual failed test cases. We observe that the scores on this simpler approach give a statistically significantly lower performance than using XAI4FL which scores are based on the individual failed test case instances. These observations highlight that some failed instances may have a bigger contribution, and thus, should be prioritized in fault localization. This finding provides one direction for future work, which is to update existing tool to provide developers with ways to determine the failed test cases that should be considered more important. One possible approach is to allow developers to input the weights of failed test cases, where the more important test cases have a bigger weight. Using this approach, developers can specify which part of the test suite they want to focus on. With this additional input, the fault localization tool may provide a more accurate prediction that is more directed towards the failed test cases that are prioritized.

Excessive number of features may reduce performance. Too many features (in our case: number of lines of code) can make the model perform badly. As an example, consider the performance of XAI4FL in *Closure* project (c.f. row 3 of Table 9). Compared to the results on other projects, *Closure* project results are comparatively low, especially on Top-5, Top-10, and *EXAM* scores. Upon further analysis, we find that the number of statements/feature in *Closure* project are much higher compared to the other projects. *Closure* project has 16,538 features, while other projects have a number of features that range from 850 to 5,150. This indicates that the bigger number of features may lower the performance of fault localization. One possible solution to address this problem is to utilize the results of SBFL in combination with XAI4FL. By applying SBFL first, we can retrieve the Top-K (e.g., K=200) predictions, which are then used as the features for XAI4FL. This is based on the intuition that by reducing the number of features, we would make the feature denser, which will enable better predictions from the model. As an illustration, we try to initialize XAI4FL using only the 200 top statements produced by SBFL. We run this experiment using the faults of *Closure* project that is successfully localized in Top-200 by SBFL (i.e., 91 faults). The result of this experiment is shown in Table 10. We observe that the combination of XAI4FL and the SBFL technique achieves better performance compared to using only XAI4FL. This combination is one possibility for future work.

Table 10: Top-K and *EXAM* scores for 91 faults from Closure project of XAI4FL max variant and XAI4FL + DStar. Results in bold is the best for each category.

Technique	Top-1	Top-5	Top-10	*EXAM* score
XAI4FL (Max variant)	6.0% (8)	15.0% (20)	24.1% (32)	0.01135
XAI4FL + DStar	**9% (12)**	**20.3% (27)**	**28.6% (38)**	**0.00154**

7 THREATS TO VALIDITY

Internal validity. A source of threat to internal validity is the correctness of the implementation of our approach. We have checked our implementation to ensure its correctness, and we utilize scikit-learn[2], one of the most popular machine learning libraries to build the classifier. For the implementation of the model-agnostic explanation technique, we utilize the SHAP library that has been used in 3,501 projects in GitHub and has 14.5k stars. We also open-source our code so other researchers can validate our findings. Thus, we believe that the threat is minimal.

External validity. A source of threat to external validity is the generalizability of our approach. We implement our XAI4FL approach and compare its performance with other SBFL techniques. For this comparison, we utilize version 1.2.0 of Defects4J dataset [18], a dataset of real faults collected from six Java projects. It is possible that we may find differing results in other settings (e.g., different dataset, different programming language, etc.). However, as the

[2]https://scikit-learn.org/stable/

Defects4J dataset that we use has been commonly used to benchmark the performance of fault localization techniques in many prior studies [21, 31], we believe that this threat is minimal.

Construct validity. A source of threat to construct validity is the metrics that we use for our evaluation. We evaluate and compare the performance of our XAI4FL against other SBFL techniques by using Top-K [23, 31] and *EXAM* score [17, 24, 31, 41] metrics. Specifically, we focus our evaluation on the Top-1 metric. Many automated program repair tools assume perfect fault localization [9, 14, 27], which makes it crucial for fault localization to identify the fault in the Top-1 result. There may be other metrics that are also suitable for this performance comparison. Nevertheless, as both the Top-K and EXAM scores are often used in many previous studies [17, 23, 24, 31, 41], there is minimal threat to this potential issue.

8 RELATED WORK

Besides the SBFL techniques [41] introduced in Section 2, more works on fault localization have been proposed [6, 22, 33, 34, 49]. Work by Roychowdhury and Khurshid [33] modeled fault localization task as a feature selection. Le et al. [23] extended the previous work by adding constraints on feature selection rather than directly using the scores computed by a feature selection. Another work by Le et al. [6] attempted to reduce the debugging effort for the developer by proposing fault localization techniques that utilize learning-to-rank strategy (i.e., inferred variant) which is used to rank faulty method based on the likelihood whether it is a root cause to a failure. Sohn et al. [34] extended SBFL by having more input called code and change metrics such as size, age, and code churn. They combined the suspiciousness score from SBFL formula and the code and change metrics as a feature for the machine learning algorithm. Zhang et al. [49] enhanced the SBFL techniques by utilizing the PageRank algorithm to recalculate the spectra information by taking into account the contributions of different tests. Using this approach, they can run the traditional SBFL techniques using the updated spectra information. Meanwhile, Laghari et al. [22] extended SBFL by leveraging the hit spectrum with frequent item mining. In our study, we utilize eXplainable Artificial Intelligence (XAI) for statement-level fault localization by implementing XAI4FL. We compare our XAI4FL with popular SBFL techniques that are commonly used in recent previous studies [31, 37, 47], which are DStar, Barinel, Ochiai, Tarantula, and O^P.

As for the usage of model-agnostic approach in software engineering tasks, there have been several papers [15, 19, 36] that investigate the usage of model-agnostic for defects prediction. Different from fault localization which is used to find the location of program units that cause a failure based on test data, defect prediction is used to find potential buggy program units based on historical data. They are also deployed in different phases of software development. Defect prediction is used to predict whether a defect may appear in the future. Meanwhile, SBFL is used to locate faults based on the results of test cases (i.e., failed and passed test cases). Work by Jiarpakdee et al. [15] focused on an empirical study by evaluating model-agnostic explanation technique in defect prediction task. They utilized 65 software metrics (e.g., number of lines added, code changes, number of owner, etc.) as the inputs of their defect prediction model. Meanwhile, Khanan et al. [19], built a defect prediction framework that uses model agnostic explanation technique to give feedback for the developers. The model-agnostic explanation technique is based on the empirical study by Jiarpakdee et al. [15]. This tool is already integrated with Github CI/CD pipeline to make it easier for developers to use the tool. Wattanakriengkrai et al. [36], proposed defect prediction in line-level by utilizing model-agnostic explanation technique. Unlike other previous studies [15, 19] that used model-agnostic explanation technique to justify the results of defect prediction to the developer, they leverage model-agnostic to get better performance and more-fine grained results. This study is one of the examples of the utilization of the XAI technique to improve the prediction of the model. To the best of our knowledge, there has been no study which utilizes model-agnostic explanation technique for fault localization.

9 CONCLUSION AND FUTURE DIRECTIONS

The research on eXplainable Artificial Intelligence (XAI) has been gaining interest recently, mainly to improve the transparency and actionable of AI solutions. In this paper, we explore another usage of XAI for enhancing the effectiveness of spectrum-based fault localization (SBFL) by proposing XAI4FL, the first XAI-supported SBFL approach. XAI4FL models SBFL as a classification task, in which program spectra of each test case are used to predict the failure/pass of the test case. XAI4FL applies XAI to learn the local importance of each program unit to each test result.

We build three variants of XAI4FL based on how we aggregate the feature importance scores for each failed test case. The results show that *max case* shows the best performance while the differences are statistically insignificant compared to the two other variants (i.e., *min case* and *mean case*). We compare the XAI4FL with the simpler approach to get feature importance in the tree-based model (i.e., MDI). The XAI4FL results statistically significantly outperform the results from MDI approach. We also compare XAI4FL with five popular traditional SBFL approaches (i.e., DStar, Tarantula, Barinel, O^P, and Ochiai) on the Defects4J dataset. Our results show that the number of real faults successfully localized at Top-1 using XAI4FL is more than twice that is localized by DStar for most projects (four out of six). As the previous studies [9, 14, 27] in automated program repair assumes perfect fault localization (i.e., fault localized in Top-1). This shows that improvements for the Top-1 results are deemed as needed and important.

For future work, we plan to improve XAI4FL by exploring other XAI techniques and designing a new XAI model for FL while considering the program spectra characteristics. We also plan to extend our evaluation by considering more baselines and projects containing real faults. Our replication package is available at https://github.com/soarsmu/XAI4FL.

Acknowledgement. This research / project is supported by the National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), with funding reference number: RGPIN-2019-05071.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE. https://doi.org/10.1109/PRDC.2006.18

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE. https://doi.org/10.1109/ASE.2009.25

[3] Anita S Acharya, Anupam Prakash, Pikee Saxena, and Aruna Nigam. 2013. Sampling: Why and how of it. *Indian Journal of Medical Specialties* 4, 2 (2013), 330–333.

[4] Amina Adadi and Mohammed Berrada. 2018. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access* 6 (2018). https://doi.org/10.1109/ACCESS.2018.2870052

[5] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2011. Fault localization for dynamic web applications. *IEEE Transactions on Software Engineering* 38, 2 (2011).

[6] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 177–188. https://doi.org/10.1145/2931037.2931049

[7] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001).

[8] Vincent Charles, Juan Aparicio, and Joe Zhu. 2019. The curse of dimensionality of decision-making units: A simple approach to increase the discriminatory power of data envelopment analysis. *European Journal of Operational Research* 279, 3 (2019), 929–940.

[9] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).

[10] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993).

[11] Arun Das and Paul Rad. 2020. Opportunities and challenges in explainable artificial intelligence (xai): A survey. *arXiv preprint arXiv:2006.11371* (2020).

[12] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3293882.3330559

[13] Melinda R Hess and Jeffrey D Kromrey. 2004. Robust confidence intervals for effect sizes: A comparative study of Cohen'sd and Cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the AERA*. Citeseer.

[14] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE.

[15] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Hoa Khanh Dam, and John Grundy. 2020. An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering* (2020).

[16] James A Jones, Mary Jean Harrold, and John T Stasko. 2001. Visualization for fault localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Citeseer.

[17] Xiaolin Ju, Shujuan Jiang, Xiang Chen, Xingya Wang, Yanmei Zhang, and Heling Cao. 2014. HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software* 90 (2014). https://doi.org/10.1016/j.jss.2013.11.1109

[18] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/2610384.2628055

[19] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. 2020. JITBot: an explainable just-in-time defect prediction bot. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*.

[20] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/2931037.2931051

[21] Yiğit Küçük, Tim AD Henderson, and Andy Podgurski. 2019. The impact of rare failures on statistical fault localization: the case of the defects4j suite. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

[22] Gulsher Laghari and Serge Demeyer. 2018. On the use of sequence mining within spectrum based fault localisation. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 1916–1924. https://doi.org/10.1145/3167132.3167337

[23] Tien-Duy B Le, David Lo, and Ming Li. 2015. Constrained feature selection for localizing faults. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

[24] Tien-Duy B. Le, David Lo, and Ferdian Thung. 2015. Should I Follow This Fault Localization Tool's Output? *Empirical Software Engineering* 20, 5 (Oct. 2015). https://doi.org/10.1007/s10664-014-9349-1

[25] Lucia, David Lo, and Xin Xia. 2014. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. https://doi.org/10.1145/2642937.2642983

[26] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st international conference on neural information processing systems*.

[27] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT ISSTA*.

[28] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011).

[29] Stefano Nembrini, Inke R König, and Marvin N Wright. 2018. The revival of the Gini importance? *Bioinformatics* 34, 21 (05 2018), 3711–3718. https://doi.org/10.1093/bioinformatics/bty373 arXiv:https://academic.oup.com/bioinformatics/article-pdf/34/21/3711/26146978/bty373.pdf

[30] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*.

[31] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. 2017. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*.

[32] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. https://doi.org/10.1145/2939672.2939778

[33] Shounak Roychowdhury and Sarfraz Khurshid. 2011. Software fault localization using feature selection. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*.

[34] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.

[35] Laura Toloşi and Thomas Lengauer. 2011. Classification with correlated features: unreliability of feature ranking and solutions. *Bioinformatics* 27, 14 (05 2011), 1986–1994. https://doi.org/10.1093/bioinformatics/btr300 arXiv:https://academic.oup.com/bioinformatics/article-pdf/27/14/1986/18530216/btr300.pdf

[36] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2021. Predicting defective lines using a model-agnostic technique. *2021 IEEE/ACM International Conference on Software Engineering (ICSE)* (2021).

[37] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE Transactions on Software Engineering (TSE)* 47, 11 (2021), 2348–2368. https://doi.org/10.1109/TSE.2019.2948158

[38] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th ICSE*. IEEE.

[39] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer.

[40] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. 2008. A crosstab-based statistical method for effective fault localization. In *2008 1st international conference on software testing, verification, and validation*. IEEE, 42–51.

[41] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013).

[42] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering (TSE)* 42, 8 (2016).

[43] Craig S Wright and Tanveer A Zia. 2011. A quantitative analysis into the economics of correcting software bugs. In *Computational Intelligence in Security for Information Systems*. Springer.

[44] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. 2016. "automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

[45] Yan Xiaobo, Liu Bin, and Wang Shihai. 2021. A Test Restoration Method based on Genetic Algorithm for effective fault localization in multiple-fault programs. *Journal of Systems and Software* 172 (2021), 110861. https://doi.org/10.1016/j.jss.2020.110861

[46] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault

localization. *ACM Transactions on software engineering and methodology (TOSEM)* 22, 4 (2013).

[47] Deheng Yang, Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2021. Evaluating the usage of fault localization in automated program repair: an empirical study. *Frontiers of Computer Science* 15, 1 (2021), 1–15.

[48] Shin Yoo, Mark Harman, and David Clark. 2013. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on software engineering and methodology (TOSEM)* 22, 3 (2013).

[49] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–272.