Intelligent Automation & Soft Computing
DOI:10.32604/iasc.2022.021163
Article

Tech Science Press

A Learning-Based Fault Localization Approach Using Subset of Likely and Dynamic Invariants

Asadullah Shaikh[1,*], **Syed Rizwan**[2], **Abdullah Alghamdi**[1], **Noman Islam**[2], **M.A. Elmagzoub**[1] **and Darakhshan Syed**[2]

[1]College of Computer Science and Information Systems, Najran University, Najran, 61441, Saudi Arabia
[2]Computer Science Department, Iqra University, Karachi, 75500, Pakistan
*Corresponding Author: Asadullah Shaikh. Email: shaikhasad@hotmail.com
Received: 25 June 2021; Accepted: 26 July 2021

Abstract: Fault localization is one of the main tasks of software debugging. Developers spend a lot of time, cost, and effort to locate the faults correctly manually. For reducing this effort, many automatic fault localization techniques have been proposed, which inputs test suites and outputs a sorted list of faulty entities of the program. For further enhancement in this area, we developed a system called SILearning, which is based on invariant analysis. It learns from some existing fixed bugs to locate faulty methods in the program. It combines machine-learned ranking, program invariant differences, and spectrum-based fault localization (SBFL). Using the execution of test cases and code coverage analysis, it obtains each method's invariant differences and the suspiciousness value based on the program spectral location and uses them as features for ranking the faulty methods. The experimental analysis of SILearning has been performed on the dataset of real fault which is extracted from the database Defects4J. The tools used in this research are Daikon and cobertura for detection of the invariants and code coverage, respectively. The results show that SILearning performs better when combined features are utilized and can successfully locate the faulty methods on average for 76.1, 90.4, 108.2, 123, and 143.5 at the top positions of 1, 2, 3, 4, and 5.

Keywords: Automatic fault localization; program invariant; spectrum-based fault localization; learning to rank

1 Introduction

The software has turned into a consistent need in our daily lives by providing a variety of services such as word processing, ATM, mobile applications, navigation systems, etc. As the product's demand is constantly increasing, the management forces the developer to release the versions by adding new features rapidly. Due to this force, the software will certainly have bugs and create issues. To fix such bugs manually, one of the fundamental responsibilities is to recognize those program entities associated as buggy and then settle those by doing some fundamental changes, which is a tedious and costly process. Thus, the demand for automated techniques is high, which helps developers to debug with minimal human intervention. Automated debugging approaches are likely to direct the support of developers in recent work of automated program repair [1].

These techniques utilize automated debugging approaches as an initial phase to recognize those elements that look faulty and provide the lists that guide program repair tools to create the program's patches that improve the bug and lead the failing test cases to pass. Therefore, in the effectiveness of program repair tools, the exactness of automated debugging approaches plays a significant part. Consequently, there is a necessity to enhance the viability of automated debugging tools to more help the programmers and the current techniques of program repair.

In this work, we worked on the family of automated debugging solutions that takes passing and failing test cases as input. Passing test cases are those test cases if their actual result and expected result match otherwise fail. Then it indicates those program entities that are faulty and probably the root cause of the failures (failing test cases). Even though these strategies have been demonstrated successfully in numerous specific situations, but there is a need to be more enhanced in the efficiency of localizing different faults. To improve this direction, we propose a system called SILearning, which uses a technique of machine learning called learning-to-rank that recognizes those methods which are faulty and ranks them based on their suspiciousness. Doing this examines three kinds of features: code coverage analysis, a score of suspiciousness which is calculated by SBFL, and the difference of subset of likely and dynamic invariants that are examined by running on both test cases (failing and passing).

To achieve the above-mentioned objectives, we have divided the system into five major steps: a cluster of faulty methods and subset selection of test cases, likely and dynamic invariant detection, subset selection of invariants, feature extraction, and method ranking. After accomplishing all these five steps, it generates the list in which the faulty methods are ranked in light of their suspiciousness. To summarize the work, the paper investigates the following research questions [2]:

- *How effectual is SILearning?* For this question, we analyze how sufficiently SILearning diagnoses faulty methods for the 357 bugs by computing average slb@n (count the number of successfully localized bugs), uef@n (useful effort), and MAP scores ($n \in \{1, 2, 3\ldots 10\}$).
- *How does it split the data into training and testing to work fruitfully?* In statistics and machine learning, we make subsets of data after splitting it into training and testing data to make predictions. When we do this, sometimes there will be a problem of underfitting or overfitting, which affects the predictability, so we have to split our data like this not to make this problem.
- *Which feature sets have a significant impact on performance?* Feature extraction is a transformation of input data into feature sets. Features are different characteristics of input data that help distinguish between other classes. In our research, all features are utilized from code coverage, invariant difference, and suspiciousness score. Then for the comparison, we also evaluate the contribution independently.
- *What is the efficiency of SILearning?* To observe the efficiency, we calculate the running time of SILearning to produce a list in which buggy methods are ranked and check their reliability.

The rest of the paper is organized as follows: Section 1 is introductory, explaining the topic's research background, significance, and research questions. In Section 2, the literature review is introduced to present the experience on fault localization. Section 3 demonstrates an architecture of SILearning which motivates from the methodology of Savant [2]. In Section 4, the methodology of the system is elaborated. We describe our experimental setup in Section 5. Section 6 presents the discussions related to each research question. Finally, Section 7 provides conclusions and future directions.

2 Literature Review

Different researchers have done different experimental investigations in fault localization to keep its functionality more effective.

Parnin et al. [3] used program slicing and the Tarantula technique to examine how programmers utilize and take advantage of automated debugging tools through an arrangement of human examinations. Furthermore, Bandyopadhyay et al. [4] enhance the effectualness of spectrum-based fault localization by including the comparative significance of various test cases in the computation of suspiciousness scores. Moreover, this technique performs better than Qchiai.

Alves et al. [5] show that spectrum-based fault localization is improved and more effective by using dynamic slicing. Sanchez et al. [6] presented an algorithm of test prioritization for fault localization called RAPTOR. This algorithm was relied on decreasing the resemblance between the patterns of statement execution as the testing progress.

Delahaye et al. [7] proposed a new technique to enhance fault localization accuracy. This technique's name is μTIL, which relies on explaining constraint and a statistical testing method of mutation-based. Furthermore, Zhou et al. [8] present a method called BugLocator, which is based on IR. It utilizes the revised Vector Space model (rVSM) to localize the files relied on initial bug reports. Moreover, Gong et al. [9] present a strategy to select the test cases in light of Diversity Maximization Speedup (DMS). This approach is utilized to maximize effectiveness.

Sahoo et al. [10] presented the automatic diagnostic strategy for identifying the main reason for software failures by utilizing the likely program invariants. Furthermore, in the same year, Qi et al. [11] presented a new direction of research to make automated fault localization techniques from the perspective of fully automated debugging. They also present metrics called NCP scores to evaluate and differentiate the effectualness of a different technique. Moreover, a two-phase prediction model was used for debugging by Kim et al. [12] for debugging. It utilizes the contents of bug reports to recommend the files expected to be fixed. Saha et al. [13] proposed a tool for automatic fault localization called BLUiR (Bug Localization Using information Retrieval). It only needs the bug reports and source code, and if there is a resemblance in data, it uses that data to localize the bug.

Just et al. [14] introduced Defects4J, an accumulation or database of reproducible real and isolated bugs and a supporting framework with the primary objective to advance the research in software engineering. Furthermore, Xuan et al. [15] proposed a learning-based approach called MULTRIC, which is utilized to merge several ranking metrics.

Wang et al. [16] IR (Information Retrieval) based Technique was used. Furthermore, to make them more efficient and better work, Le et al. [17] combined the two techniques spectrum based technique and IR based approach, and named as Multimodal bug Localization (AML). The empirical study shows that AML outperforms a single technique.

Le et al. [2] proposed a Savant system, which is a rank-based approach using likely invariants.

Sohn et al. [18] extended SBFL using code and change metrics in the context of defect prediction using real-world faults. Furthermore, Ang et al. [19] revisited the research direction of automated fault localization and recommend the SFL research community to focus on creating an ecosystem that can be used by developers during debugging.

Zhang et al. [20] proposed an approach that worked on test classification to enable the use of unlabelled test cases in localizing faults. Gao et al. [21] proposed a framework of multiple fault localization based on machine learning and identified the Random forest algorithm as the most efficient algorithm to identify the position of bugs.

Zou et al. [22] revealed that combined technique significantly outperforms any individual technique and suggested that combination may be a desirable way to apply fault localization technique. Wang et al. [23] analyzed the role of optimization techniques over the simple invariant-based method. Maru et al. [24] proposed an effective approach for fault localization based on a back-propagation neural network that utilizes branch and function coverage information along with test case execution results to train the network.

Maru et al. [25] utilized a back propagation neural network in which the actual number of times the statement is executed to train the network and got a 35% increase in the effectiveness over existing BPNN. Zakari et al. [26] reviewed existing research on Multiple fault localization (MFL) in software fault localization. Ghosh et al. [27] used a framework based on deep learning which was able to calculate the suspicious score of each statement and rank accordingly.

The notion of a required location set is defined by Lahiri et al. [28] in a research study. Conceptually, such a set should contain at least one program location from each issue fix. As a result, it's difficult to solve the bug without modifying at least one of these locations. If a fault localization approach delivers a must location set for each and every defective program and every flaw in the program, it is named a must algorithm. The concept of necessary fault localization, according to observations, is dependent on the improvement scheme adopted, which determines the steps that can be taken to program statements as part of a correction. As a result, a new technique for defect localization has been created, and its necessity has been demonstrated in comparison to regularly used strategies in automated program repairing. The findings suggest that problem localization can greatly speed up the repair process while not sacrificing any possible restorations.

By employing an assessment-based structured methodology, Zakari et al. [29] propose the method of analysis for multiple fault localization. This study found 55 studies that applied to four research questions. The methodology includes a systematic design and evaluation of procedure that is based on reliable and reproducible evidence-based analyses. Although this technique seems to have become increasingly problematic, current techniques in the sector are still in their infancy. Few studies use real flaws in their trials. There are fewer concrete options to minimize multiple fault localization debugging time and cost by using a technique like MBA debugging, which necessitates more attention among the researchers.

For the first time, Lou et al. [30] suggest a unified debugging technique to unite the two regions in the opposite direction, i.e., can program repair aid with fault localization? Not only does this research open up a new area for more efficient fault localization, but it also expands the range of program repair to include all conceivable flaws. Through both unsupervised and supervised learning, this method aims to enhance state-of-the-art defect localization.

Zhang et al. [31] investigate the use of dataset resampling to mitigate the negative effects of the imbalanced class dilemma and improves the precision of the deep-learning-based fault localization proposed model in this study. Deep-learning-based fault localization, in particular, may necessitate duplicate vital data to improve the weaker but useful experience caused by the imbalanced class collections. This study uses the passing/failing property of test scenarios to recognize failed test cases as redundant essential data and proposes an adaptive oversampling strategy to resample failed test cases in order to produce a class fair test suite.

Peng et al. [32] propose an Autoencoder Based Practical Strategy for Fault Localization to increase accuracy and practicality in fault code localization. It starts with an autoencoder that extracts 32 properties from static source code in applications. The software then uses Spectrum Based Fault Localization methods to generate 14 other types of ratings, which are used as an additional set of features during software execution.

DeepRL4FL is a deep learning fault localization solution suggested by Li et al. [33], which treats fault localization as an image pattern recognition problem and detects erroneous code at the expression and method layers. DeepRL4FL achieves this via unique code coverage classification tasks and data requirements representation learning for program execution.

Bartocci et al. [34] present a method for combining testing, description mining, and fault diagnosis in a novel cyber-physical system called Debug. To illustrate errors in Simulation models dynamically. This research addresses the hybrid character of cyber-physical systems in particular by employing a variety of methods to infer characteristics from the model's discrete and continuous system parameters. The research put debugger to the test in two methods including two primary scenarios and several types of errors, proving the approach's future benefits.

Assi et al. [35] conducted research to determine the effects of fortuitous accuracy, in both strong and weak variants, on the efficacy of publicity test suite reduction, Test Case Prioritization, and Spectrum-based Fault Localization.

Wardat et al. [36] present a method for discovering root causes that includes collecting any numerical inaccuracy, evaluating the model throughout training, and determining the impact of each component on the DNNeep neural network output. Researchers compiled a test from Stack Overflow and GitHub that includes 40 problematic algorithms and patches with real faults in deep learning techniques.

The benchmark utilized in this research can be used to assess autonomous debugging tools and repair approaches. This deep learning bug-and-patch benchmark is used to assess this method. The findings indicate that this methodology is far more efficient than the current debugging method utilized in the Keras library's current state-of-the-art.

In their investigation on fault localization, Zhang et al. [37] use three representative deep learning architectures. Convolutional neural networks, recurrent neural networks, and multi-layer perceptrons are examples of these structures. This research used large-scale experimentation on 8 real-world programs with all glaring weaknesses to see how efficient they were at fault localization. Among the analyzed designs, the convolutional neural network is the most successful for fault localization in the scenario of genuine faults, and researchers identify potential deep learning parameters for increasing fault localization.

Based on the literature review (summarized in Tab. 1), it can be seen that several novel approaches have been proposed in the literature. The dominant approach is using machine learning for fault identification/ localization. However, only a few papers have investigated that subset of likely and dynamic invariants, and the idea presented in this paper can be regarded as a novel approach in this area. The next section presents the details of this approach.

Table 1: Summary of literature on fault localization

Paper	Year	Techniques
Ghosh et al. [27]	2021	Deep Learning
Zakari et al. [29]	2021	Assessment based approach
Lou et al. [30]	2020	Supervised/unsupervised learning
Zhang et al. [31]	2021	Oversampling, Neural networks
Peng et al. [32]	2020	Auto-encoders
Li et al. [33]	2021	Image pattern recognition
Bartocci et al. [34]	2021	Cyber physical systems
Assi et al. [35]	2021	Spectrum based approach
Wardat et al. [36]	2021	MLP
Zhang et al. [37]	2021	CNN, RNN, MLP

3 System Overview and Background

Fig. 1 demonstrates the architecture of SILearning, which includes five stages: cluster of faulty methods and subset selection of test cases, likely and dynamic invariant detection, subset selection of invariants, feature extraction, and method ranking. In the training phase, it uses a machine learning model to arrange the faulty methods in order based on the likelihood of failure. In the phase of deployment, first, it takes the ranked model which is learned in the phase of training, and then outputs an ordered list of methods which are the original reason for the failing experiments.

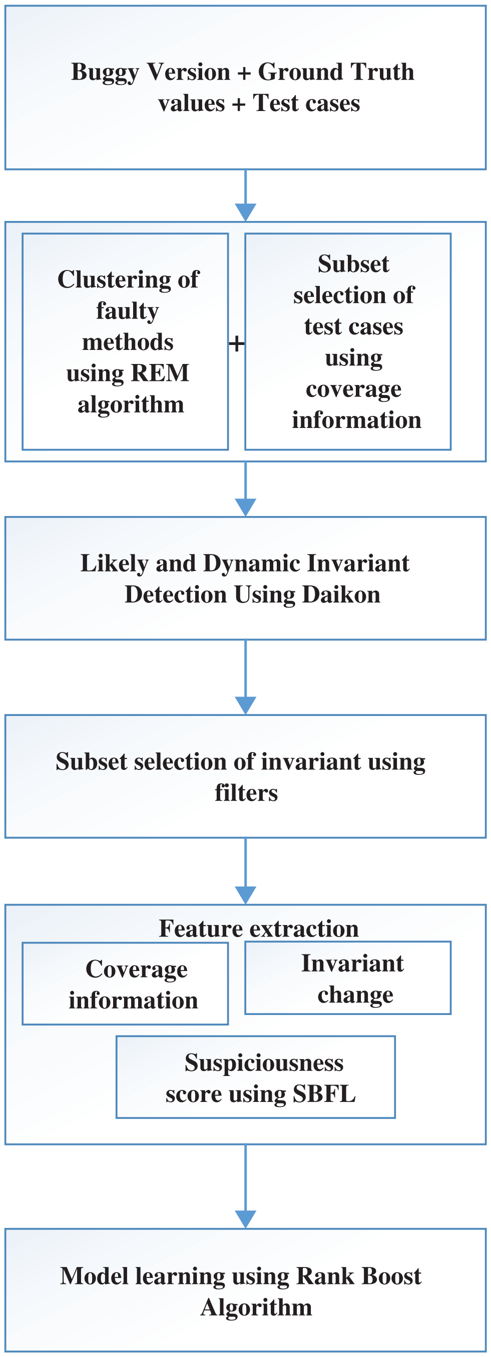
Buggy Version + Ground Truth values + Test cases
Clustering of faulty methods using REM algorithm
+
Subset selection of test cases using coverage information
Likely and Dynamic Invariant Detection Using Daikon
Subset selection of invariant using filters
Feature extraction
Coverage information
Invariant change
Suspiciousness score using SBFL
Model learning using Rank Boost Algorithm

Figure 1: Architecture of SILearning

3.1 Cluster of Faulty Methods and Subset Selection of Test Cases

The faulty program is obligatory to keep running on both test cases (passing and failing) so that execution traces will be recorded to inference the invariant that can be too costly. The cost of detection of invariants depends on the number of test cases and instrumented method as shown in Fig. 2.

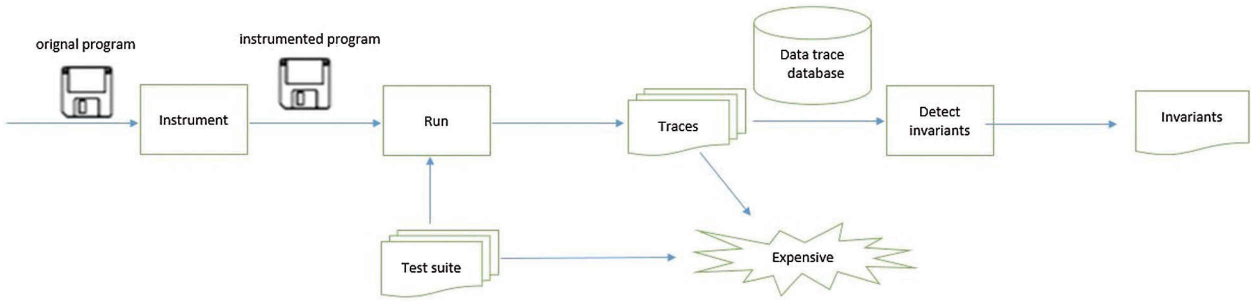

Figure 2: Detection of invariant with different test suits

By doing this step, we get a set of clusters of methods; every cluster of methods has a related subset of failing test suits and passing test suits. The number of clusters depends on various factors such as the size of clusters and users' preferences. The main objective of this step is to make the technique faster by decreasing the runtime cost and limiting the memory.

3.2 Likely and Dynamic Invariant Detection

Likely and dynamic program invariants are reported by Ernst et al. [38]. It doesn't utilize the source code while gathering invariants. It does not require any software engineer to give comments even though it acknowledges the directions and creates the documentation as a record when the code is instrumented. However, a huge test suite is needed.

3.3 Subset Selection of Invariant

Their susceptibility constrains the actual use of invariants for localizing the anomaly to the number and nature of the system's examined variables. However, it uncovers many invariants, and just a subset of them steadily catches its exact behavior, while a large fragment is either futile or very volatile. We settle this issue by choosing those invariants, which aim for buggy programs and are useful in fault localization.

3.4 Extraction of Features

There are different kinds of features used in fault localization. The features we use in SILearning are spectrum-based fault localization, likely and dynamic invariants, and some metrics.

3.4.1 Spectrum-Based Fault Localization

The objective of spectrum-based fault localization (SBFL) is to pinpoint program entities that are faulty and potentially rank them in light of failing and passing test case execution. There can be other ways of ranking such as based on static and dynamic features. A typical instinct is that those entities of a program implemented more frequently by failed test suits yet never or once in a while through passing test cases are more probably to be faulty. Fig. 3 shows the overview of SBFL.

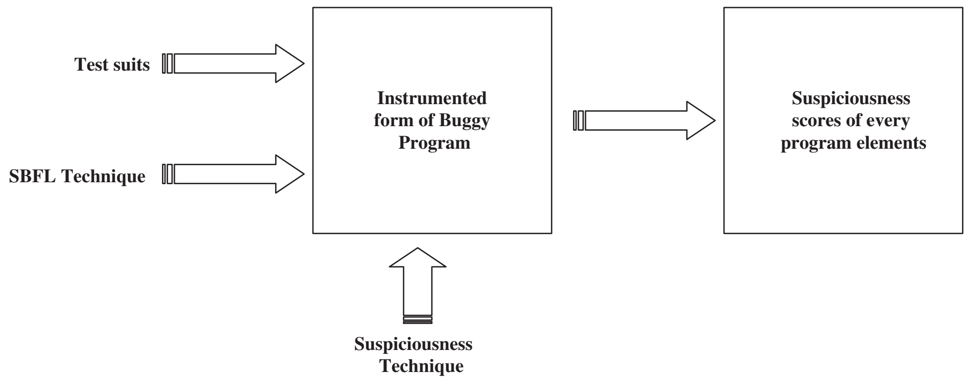
Test suits
SBFL Technique
Instrumented form of Buggy Program
Suspiciousness scores of every program elements
Suspiciousness Technique

Figure 3: Overview of SBFL

The input of SBFL technique is a version of a buggy program and an arrangement of passing and failing test suits. When test cases are run on an instrumented form of a faulty program, it generates program spectra relating to an arrangement of program entities implemented by every test case. Then SBFL algorithms calculate an arrangement of measurement for distinguishing every component "c" of a program that shows up in a spectrum. A set of statistics is shown in Tab. 2. These measurements notify the calculation of the suspiciousness score for every programming entity. SBFL algorithms differ in the fundamental formula [38].

Table 2: Statistic symbol with explanation for SBFL [2]

Symbol	Explanation
f (c)	Number of failing test cases that execute program element e
f (c′)	Number of failing test cases that do not execute program element e
p(c)	Number of passing test cases that execute program element e
p(c′)	Number of passing test cases that do not execute program element e
F	Total number of failing test cases
P	Total number of passing test cases

For figuring the suspiciousness scores of program entities, many researchers have been researched and proposed a wide assortment of mathematical models in light of unsuccessful and successful test suits.

3.4.2 Invariant Changes

Change of invariants is the change in program state or condition at a specific program point, which differentiate across various versions of a program. The progression happens in technique invariants deduced from multiple arrangements of execution traces that convert one invariant into other. Those places or locations are faulty where these invariants changes happen.

3.5 Learning to Rank

Learning to rank is one of the well-known machine learning applications during the development of ranking models for data recovery frameworks. The main objective is to deduce an ideal method for consolidating the features that decrease the loss function and outputs a list of ranked and sorted documents by their computed relevance to the input inquiries.

4 Methodology of the System

4.1 Cluster of Faulty Methods and Subset Selection of Test Cases

Algorithm 1: Cluster of faulty methods and subset selection of test cases

```
Input:M: all methods executed by failing test cases
      Sci, USci: successful and unsuccessful test cases, respectively
       S: maximum size of the cluster
       T: minimum acceptable coverage
Output: Cluster of methods and associated test suites
let $C_s$ ←reduce_size(rem(M,$\frac{M}{S}$))
let $C_r$ ← empty
foreach $c_m \in$ Cs do
      let $Sc_i^s$ ← code_coverage($Sc_i$, $c_m$)
      let$Sc_c$← empty
      let $USc_i^s$ ← code_coverage($USc_i$, $c_m$)
      let $USc_c$ ← empty
      while$\exists m \in c$ s.t. coverage(m,$Sc_c$) < M do
            let t ← pop($Sc_i^s$)
            if t covers at least one method m $\in$c then
                  $Sc_c \leftarrow Sc_c \cup \{t\}$

            end
      end
      while $\exists$ m $\in$c s.t. coverage(m, $USc_c$) < M do
            ifUSc has approximately same coverage as others then
                  $US_c$ ← rem ($USc_c$,$\frac{USc}{S}$)
            end
      end
      $C_r \leftarrow C_r \cup \{c_m, Sc_c\} \cup \{USc_c\}$
end
return $C_r$
```

To record execution traces for invariant induction, the defective program must be run on unsuccessful and successful test cases by SILearning. Both processes (mining the invariants and collection of traces) can be too costly for medium to large projects because the project's size is directly proportional to the quantity of executed test suits and methods, which mutually add the expense in debugging. When all the methods are executed by all test cases, the runtime cost of Daikon to derive invariants is high. Meanwhile, we should gather sufficient information to help exact and valuable invariant derivation.

We overcome this problem to make an optimization technique named "cluster of faulty methods and subset selection of test cases." It enhances the capability to don't use all the methods and test cases for invariant derivation on extensive projects with the end goal of fault localization. In the first place, we eliminate all non-faulty methods. Second, we cluster all those faulty methods which are implemented on successful test cases (passing). Then separately for each cluster record, the execution traces by running unsuccessful test cases and afterward, chose the successful test cases as subset because referring every successful test case is unimportant and wastage of time and resources. Then for the unsuccessful test cases, we group the unsuccessful test cases as a cluster that has the same code coverage value and utilize these clusters of unsuccessful test cases accordingly.

Algorithm1 represents this step's functionality called "cluster of faulty methods and subset selection of test cases". First, it analyzes each faulty method's code coverage and then constitutes each faulty method through a code coverage vector depicting all input passing test cases. If the corresponding test case covers the method, then it has one value otherwise zero. SILearning utilizes the Robust Expectation Maximization clustering algorithm [39] to similar group methods that have the same value. It is an improved version of the expectation-maximization algorithm [39], which is a well-known clustering method for the mixture likelihood approach. REM takes as input a faulty method with its value (coverage vector) and produces clusters based on probability where each cluster has a method in random quantity. We also fixed the maximum size of cluster "S" so that the resulting cluster split into smaller groups. We continue choosing S methods arbitrarily to make cluster new groups up to the cluster's size is decreased to less than M.

It chooses the successful test cases as a subset for every cluster that executed its methods at least T time after clustering methods. Then it uses REM algorithm to cluster the failed test cases according to their coverage value and utilize them accordingly. That is the way to optimize the technique and reduced memory and time. The yield of this progression is set of clusters Cr, each having methods where there is a related subset of successful and unsuccessful test cases for each method cluster.'

4.2 Likely and Dynamic Invariant Detection

In the previous step, the cluster of faulty methods was generated, so now for every cluster, SILearning finds the information of execution for every method in the cluster over a subset of successful test cases and the clusters of unsuccessful test cases. There are three sets of test cases used by SILearning: Scc represents a subset of successful test cases. UScc shows the clusters of failed test cases and Scc U UScc is the union of the subset of successful and clusters of unsuccessful test cases. SILearning gathers the information of execution individually for the faulty methods in the clusters utilizing all three sets Scc, UScc, and (Scc U UScc). After traces of all these sets' execution, this trace information utilized is sent to the Daikon tool to gather the invariants. Daikon is a tool for the dynamic detection of likely invariants. A program is run to analyze the values that are computed by the program and then describe the characteristics that were right when the program is executed. The yield of this step is the invariants inferred by daikon, which are referred to as inv (UScc), inv (Scc), and inv (UScc∪Scc), respectively.

4.3 Subset Selection of Invariant

As the preceding step exposes the hundreds of invariants, which is very typical and challenging to analyze all the invariants for the large or medium to large projects, only some of them are valuable and stable, which reports the correct behavior of the program. After analyzing all, we recognize that blindly observing all the invariants establishes the noise and variation in the output of detection, which negatively affects the result.

We remedy this issue by selecting those invariants that are good in the target buggy program and helpful in fault localization. Hence there are different types of invariants, and every invariant has its characteristics. We analyze all the invariants' types and get to know some invariants' unimportance, so we selected those invariants to remove like "redundant invariants," which are implied by other invariants and "has only one value variable," which has the value of hashcode. We use two filters for removing these invariants: one is an "obvious filter" for removing the redundant invariant. The second is the "derived parameter filter" to remove those invariants, which are derived from the postcondition of the parameter to work only with the useful invariants.

4.4 Feature Extraction

4.4.1 Code Coverage Analysis

There are numerous instruments accessible to help in computing code coverage, which needs a specific ability to do this. In this research, the Cobertura tool is utilized, a java tool, and relied on jcoverage. It computes the percentage of code retrieved by tests and finds which areas of the program are missing in test coverage. So Cobertura checks the methods. If the methods are covered, it will check the invariant change; otherwise, not.

4.4.2 Invariant Changes

Invariant changes are the changes that occur in the invariant set between failing and passing programming execution. In this research, as we are using the Daikon to infer the invariants to find the invariant change, we use the utility of invariant diff that is constructed to give the results of differencing among the set of invariants. It estimates the similarity or dissimilarity of invariants at program points that have the same name.

4.4.3 Suspiciousness Score

Sometimes the changes to their invariants between the methods are comparable, which is more troublesome for ranking or learning models to separate amongst the defective and non-defective methods. Because of this reason, to find a fault, a suspiciousness score is also incorporated in this research as a supportive or additional feature Suspiciousness score is assigned by spectrum-based fault localization (SBFL), whose primary goal is to rank conceivable faulty program elements in light of perceptions of passing and failed test case execution. Spectrum-based fault localization utilizes statistical information such as the number of passed executed, number of failed executed, number of failed not executed, and the number of passed not executed.

SILearning calculates the suspiciousness score using statistical information calculated by the previous formulas of spectrum-based fault localization such as $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, $ER5^c$, which are proposed to do the manual calculation by Xie et al. and GP2, GP3, GP13, GP19, which are also proposed by Xie et al. [40] using an automatic genetic algorithm.

The last, the code coverage metrics, the invariant change features, and suspiciousness score features are sent to the model learning to rank the faulty methods based on these all features.

4.5 Model Learning and Method Ranking

4.5.1 Feature Scaling

Sometimes the range values of raw data that are used in the learning model vary widely. To make this data more helpful so that the objective functions will perform effectively, the large variations should be maintained. To maintain this variation, the normalization technique is needed. There are many ways to normalize the feature values between the range of zero and one, the formula which is used in SILearning is formulated as:

$$a_j^{'} = \begin{cases} 0, & a_j < min_i \\ \dfrac{a_j - min_j}{max_j - min_j}, & min_j \le a \le max_j \\ 1, & a_j > max_j \end{cases} \tag{1}$$

where a_j = original values of j^{th} feature, a_j' = normalized values of j^{th} feature, min_j = minimum value of j^{th} feature from the training data, max_j = maximum value of j^{th} feature from the training data.

4.5.2 Model Learning and Method Ranking

In this step, SILearning takes three types of input to rank the faulty methods: fixed bugs, all the features discussed in feature extraction, and the ground truth faulty methods. The developers alter the code of methods to correct the error; these methods give the ground truth.

SILearning uses this type of data as an input to RankBoost algorithm used as a learning model that ranks the faulty methods relies on such features.

In the deployment phase, the features extracted from new bugs are used to learn the statistical model. In the last, SILearning produces the output in the form of a list of ranked faulty methods generated by the learned model so that the engineers can use it for inspection.

5 Experimental Analysis

5.1 Experiment Environment

The environment under which these experiments are conducted consisted of Intel(R) Core(TM) i3-4005U CPU @1.70 GHz and 4 GB physical memory. The operating systems are Ubuntu 14.04 and Windows 10. We carried out the experiments by using python 3.

5.1.1 Data Collection

SILearning is examined on the real bugs in the Defects4J benchmark [13]. It is an accumulation of reproducible real and isolated bugs and a supporting framework with the primary objective of advancing the research in software engineering. Defects4J gives an abstraction layer for the database that facilitates the utilization of a database of bugs. Tab. 3 shows bugs in Defects4J.

Table 3: Projects and number of real bugs in Defects4J [13]

Projects	Faults	KLoc	Tests	Avg. #Methods
Closure Compiler	133	345.6	7927	7479.5
Commons Math	106	111.8	3602	4792.3
Commons Lang	65	52.6	2245	2151.1
Joda-Time	27	110.8	4130	4083.5
JFreeChart	26	132.9	2205	7782.5
Total	357	753.7	20109	26288.9

5.1.2 Cross-Validation

We use k-fold cross validation across each of five projects, which divides the dataset into k subsets and the holdout method is iterated k times. We have given a set of n faults, then we distribute the sets into n groups from which n−1 are used for training and the remaining for the testing and then iterate this practice n times by using different groups as the set for testing.

5.1.3 Metrics

We analyzed various aspects related to data such as slb@n, mue@n, and MAP. If the number of features would have been large, the correlations matrix/heat maps can also be analyzed. Also, Principal Component Analysis (PCA) or auto-encoders could have also been used. However, we restrict our discussion to the following three metrics.

5.1.3.1 slb@n

slb@n adds up the quantity of those faults which are successfully localized inside the top n positions of the produced ranked list. We utilize the absolute rank to the faults so that the developer will only monitor the top few positions in a list where the buggy statements are ranked relied on suspiciousness. We select the top 5 locations mean n = {1, 2, 3, 4, 5} and calculate the suspiciousness of slb@1, slb@2, slb@3, slb@4 and slb@5.

5.1.3.2 mue@n

mue@n estimates the energy misused at n by programmers before localizing the bug's root causes on the non-defective program. We calculate mue@n by the total quantity of non-defective methods within the top positions of the ranked list of existing methods before reaching the first faulty method. We again pick the top five positions (n = {1, 2, 3, 4, 5}) for this metric.

5.1.3.3 Mean Average Precision (MAP)

Mean Average Precision examines the methods which are ranked in information retrieval. We utilize these metrics to examine the ranked list of methods that are faulty and generated by fault localization. It is calculated as:

$$MAP = \frac{\sum_{a=1}^{N} AP}{N} \tag{2}$$

where a = rank of program elements at a position in the list, N = Total quantity of methods in the list, and AP = Average Precision which is followed as:

$$AP = \sum_{a=1}^{N} \frac{P(a) * pos(a)}{number\ of\ faulty\ methods} \tag{3}$$

where pos (a) = Boolean function shows the ath method is faulty or not and P (a) = Precision at ath position which can be calculated as:

$$P(a) = \frac{Faulty\ method\ in\ the\ top\ a}{a} \tag{4}$$

We iterated metric calculation a hundred times utilizing hundred different seeds to arbitrarily break the tie for the problem of suspiciousness of program elements.

6 Discussions

The effectualness of SILearning on faults from the five projects of the Defects4J benchmark is demonstrated in Tab. 4. From this table, we observe that out of three hundred fifty-seven (357) bugs, SILearning effectively localizes 76.1, 90.4, 108.2, 123 and 143.5 concerning the average of slb@1, slb@2, slb@3, slb@4, and slb@5, individually. We also notice the misused energy over all the projects regarding the mue@1, mue@2, mue@3, mue@4, and mue@5, which have values 234, 437.2, 718.2, 899.3, and 1069.5, and the overall Mean Average Precision score is 0.302. By comparing all the five projects, SILearning gives Lang's best effectiveness, accomplishing the most noteworthy on slb@n (31.9, 34, 39.1, 41.7, and 46.2) and a MAP score of 0.58. Over these analyses, SILearning ended Daikon once on a fault of Closure because of the time limit, and among all these five projects, Closure is prolonged a takes more time.

Table 4: Effectualness of SILearning concerning slb@n, mue@n, and MAP

Projects	Total Faults	Avg. slb@n					Avg. mue@n					Avg. MAP
		@1	@2	@3	@4	@5	@1	@2	@3	@4	@5	
Closure	133	7.0	10.1	11.4	13.1	19.2	104.1	200.1	340.9	419.4	463.2	0.10
Math	106	24.3	32.9	39.2	46.2	51.4	71.3	123.4	195.0	248.4	319.8	0.30
Lang	65	31.9	34.0	39.1	41.7	46.2	22.6	58.2	80.6	96.9	124.0	0.58
Time	27	6.3	11.1	14.5	11.6	15.7	20.7	35.0	50.9	72.7	83.5	0.28
Chart	26	6.6	2.3	4.0	10.4	11.0	15.3	20.5	50.8	61.9	79.0	0.25
Total	357	76.1	90.4	108.2	123	143.5	234	437.2	718.2	899.3	1069.5	0.302

We have to do the right setting for the cross-validation to split the data into training and testing for more effectiveness because effectiveness differs from setting to setting.

The effectualness of SILearning with different settings is demonstrated in Tab. 5. We observe that 7-fold cross-validation (k = 7) in all the settings accomplish the best execution contrasted with others. However, there is no massive difference in results, so we infer that the data measure has little effect on the accuracy of SILearning.

Table 5: Changing the size of training data concerning slb@n, mue@n and MAP

K	Avg. slb@n					Avg. mue@n					Avg. MAP
	@1	@2	@3	@4	@5	@1	@2	@3	@4	@5	
2	65.4	74.6	99.9	111.2	134.5	239	444.2	728.9	910.4	1103.4	0.291
3	66.8	77.4	102.1	113.9	136.7	241.3	449.3	731.9	913.5	1107.5	0.293
4	67.01	77.9	96.1	115.6	130.5	261.2	465.09	751.6	934.2	1139.5	0.294
5	64.9	72.1	99.5	108.4	133.2	268.3	473.2	763.05	941.4	1146.6	0.289
6	80.3	87.09	101.4	118.5	135.9	251.3	459.3	743.3	923.06	1117.4	0.301
7	81.9	99.0	106.5	119.5	139.5	228.3	432.4	715.3	890.1	1083.3	0.305
8	76.9	85.54	98.5	114.5	129.4	235.1	442.54	722.3	903.5	1095.4	0.296
9	54.07	79.4	96.4	105.4	131.02	269.1	475.07	765.5	943.2	1149.9	0.295
10	72.06	89.01	101.5	113.4	137.4	246.5	455.3	739.4	918.6	1112.65	0.297
Default	76.1	90.4	108.2	123	143.5	234	437.2	718.2	899.3	1069.5	0.302

Every feature has a different impact on the techniques based on their functionality. The effectualness of SILearning utilizing code coverage analysis, invariant changes, a score of suspiciousness, and their combinations are figured out in Tab. 6. We observe this if only one metrics is utilized, then SILearning is not more effective than the combination of them.

Table 6: Effectiveness of features for slb@n, mue@n, and MAP

Set of feature	Avg. slb@n					Avg. mue@n					Avg. MAP
	@1	@2	@3	@4	@5	@1	@2	@3	@4	@5	
Code Coverage	32.43	43.65	50.92	61.14	70.64	250.43	589.12	903.65	1031.65	1185.43	0.176
Suspiciousness score	55.32	64.65	95.24	101.45	111.34	293.34	548.08	809.16	949.76	1107.54	0.251
Invariant change	59.23	78.76	91.24	99.07	109.8	291.54	508.65	799.56	995.34	1205.67	0.213
Total	76.1	90.4	108.2	123	143.5	234	437.2	718.2	899.3	1069.5	0.302

To find the efficiency of any model, we have to check the running time on the dataset. The average running time for SILearning, including learning and ranking, is demonstrated by Tab. 7. 13.575 is the normal time with a standard deviation of 14.136 to yield a ranked list of program elements for a given bug of the projects. We notice that the Closure takes a long time to localize in all the projects as some restrictions are imposed on like comments are not preserved, etc., and the chart has the lowest average execution time.

Table 7: Executing time of SILearning

Project	Mean	Standard Deviation
Closure	29.852	3.81
Math	3.99	0.599
Lang	1.96	0.33
Time	2.5	0.47
Chart	1.525	0.283
Total	13.575	14.136

7 Conclusions and Future Work

Fault localization is an essential area in software engineering, as manually debugging consumes a lot of time and energy. The main contributions of this work are as follows:

1. A learning-based fault localization approach has been proposed, which localizes faulty programs by learning from some existing fixed bugs. It combines a machine learning approach known as "learning to rank," program invariant analysis, and spectra-based fault localization. Invariant difference and suspiciousness values are examined on both test cases and code coverage analysis as features for ranking the faulty methods in light of their probability of being a source of failure.
2. SILearning has been developed, which consists of two phases: training phase and deployment phase. Each phase has five steps: method clustering and test case selection, invariant mining, subset selection of invariant, feature extraction, method learning, and method ranking. In the last, it gives the list of faulty program elements as per the suspiciousness score.

3. We have practically analyzed the SILearning on the dataset of real faults extracted from the database "Defects4J" and done research on four questions. Our analysis shows that when SILearning works on single metrics, effectiveness is not as good as the combination. It can successfully localize the faulty methods on average for 76.1, 90.4, 108.2, 123 and 143.5 at the top positions 1, 2, 3, 4, and 5. We further analyze that by selecting the subset of invariants, there is a significant impact on the system accuracy, and more bugs can be localized quickly.

The potential implications of this work lie in the use of automated tools for spotting out bugs/ faults in code without manual intervention. Also, code correction can be performed automatically using sequence-to-sequence learning. In the future, we also plan to research the subset of invariant further and spread out the examination of SILearning by including more faults.

Funding Statement: The authors would like to express their gratitude to the ministry of education and the deanship of scientific research Najran University Kingdom of Saudi Arabia for their financial and technical support under code number NU/-/SERC/10/552.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Sym. on Principles of Programming Languages*, New York, United States, pp. 298–312, 2016.

[2] T. D. B. Le, D. Lo, C. L. Goues and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proc. of the 25th Int. Sym. on Software Testing and Analysis*, New York, United States, pp. 177–188, 2016.

[3] C. Parnin and A. Orso, "Are automated debugging techniques helping programmers?" in *Proc. of the 2011 Int. Sym. on Software Testing and Analysis*, New York, United States, pp. 199–209, 2011.

[4] A. Bandyopadhyay and S. Ghosh, "Proximity-based weighting of test cases to improve spectrum-based fault localization," in *2011 26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, Washington, United States, pp. 420–423, 2011.

[5] E. Alves, M. Gligoric, V. Jagannath and M. d'Amorim, "Fault-localization using dynamic slicing and change impact analysis," in *2011 26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, Washington, United States, pp. 520–523, 2011.

[6] A. G. Sanchez, R. Abreu, H. G. Gross and A. J. V. Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *2011 26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, Washington, United States, pp. 83–92, 2011.

[7] M. Delahaye, L. C. Briand, A. Gotlieb and M. Petit, "µTIL: Mutation-based statistical test inputs generation for automatic fault localization," in *2012 IEEE 6th Int. Conf. on Software Security and Reliability*, Washington, United States, pp. 197–206, 2012.

[8] J. Zhou, H. Zhang and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *2012 34th Int. Conf. on Software Engineering (ICSE)*, Zurich, Switzerland, pp. 14–24, 2012.

[9] L. Gong, D. Lo, L. Jiang and H. Zhang, "Diversity maximization speedup for fault localization," in *Proc. of the 27th IEEE/ACM Int. Conf. on Automated Software Engineering*, New York, United States, pp. 30–39, 2012.

[10] S. K. Sahoo, J. Criswell, C. Geigle and V. Adve, "Using likely invariants for automated software fault localization," in *Proc. of the 18th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, New York, United States, pp. 139–152, 2013.

[11] Y. Qi, X. Mao, Y. Lei and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proc. of the 2013 Int. Sym. on Software Testing and Analysis*, New York, United States, pp. 191–201, 2013.

[12] D. Kim, Y. Tao, S. Kim and A. Zeller, "Where should we fix this bug? A two-phase recommendation model," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[13] R. K. Saha, M. Lease, S. Khurshid and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, Melbourne, Australia, pp. 345–355, 2013.

[14] R. Just, D. Jalali and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. of the 2014 Int. Sym. on Software Testing and Analysis*, New York, United States, pp. 437–440, 2014.

[15] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *2014 IEEE Int. Conf. on Software Maintenance and Evolution*, British Columbia, Canada, pp. 191–200, 2014.

[16] Q. Wang, C. Parnin and A. Orso, "Evaluating the usefulness of IR-based fault localization techniques," in *Proc. of the 2015 Int. Sym. on Software Testing and Analysis*, New York, United States, pp. 1–11, 2015.

[17] T. D. B. Le, R. J. Oentaryo and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, United States, pp. 579–590, 2015.

[18] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proc. of the 26th ACM SIGSOFT Int. Sym. on Software Testing and Analysis*, New York, United States, pp. 273–283, 2017.

[19] A. Ang, A. Perez, A. Van Deursen and R. Abreu, "Revisiting the practical use of automated software fault localization techniques," in *2017 IEEE Int. Sym. on Software Reliability Engineering Workshops (ISSREW)*, Toulouse, France, pp. 175–182, 2017.

[20] X. Y. Zhang, Z. Zheng and K. Y. Cai, "Exploring the usefulness of unlabelled test cases in software fault localization," *Journal of Systems and Software*, vol. 136, no. 1, pp. 278–290, 2018.

[21] M. Gao, P. Li, C. Chen and Y. Jiang, "Research on software multiple fault localization method based on machine learning," in *MATEC Web of Conf.*, Shanghai, China, pp. 1–9, 2018.

[22] D. Zou, J. Liang, Y. Xiong, M. D. Ernst and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 332–347, 2019.

[23] T. Wang, K. Wang, X. Su and Z. Lei, "Invariant based fault localization by analyzing error propagation," *Future Generation Computer Systems*, vol. 94, no. 1, pp. 549–563, 2019.

[24] A. Maru, A. Dutta, K. V. Kumar and D. P. Mohapatra, "Software fault localization using BP neural network based on function and branch coverage," *Evolutionary Intelligence*, vol. 14, no. 1, pp. 87–104, 2019.

[25] A. Maru, A. Dutta, K. V. Kumar and D. P. Mohapatra, "Effective software fault localization using a back propagation neural network," *Computational Intelligence in Data Mining Springer*, vol. 990, no. 1, pp. 513–526, 2020.

[26] A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed and R. A. Rasheed, "Multiple fault localization of software programs: A systematic literature review," *Information and Software Technology*, vol. 124, no. 106312, pp. 1–20, 2020.

[27] D. Ghosh and J. Singh, "A novel approach of software fault prediction using deep learning technique," *Automated Software Engineering: A Deep Learning-Based Approach Springer*, vol. 8, no. 1, pp. 73–91, 2020.

[28] S. K. Lahiri and C. Wang, "Must fault localization for program repair," in *Int. Conf. on Computer Aided Verification*, Los Angeles, United States, pp. 658–680, 2020.

[29] A. Zakari, S. Lee, R. Abreu, B. Ahmed and A. Rasheed, "Multiple fault localization of software programs: A systematic literature review," *Information and Software Technology*, vol. 124, no. 1, pp. 106312, 2020.

[30] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang *et al.,* "Can automated program repair refine fault localization? A unified debugging approach," in *29th ACM SIGSOFT Int. Sym. on Software Testing and Analysis*, New York, United States, pp. 75–87, 2020.

[31] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu *et al.,* "Improving deep-learning-based fault localization with resampling," *Software: Evolution and Process*, vol. 33, no. 3, pp. e2312, 2021.

[32] Z. Peng, X. Xiao, G. Hu, S. Kumar, A. Sangaiah *et al.,* "ABFL: An autoencoder based practical approach for software fault localization," *Information Sciences*, vol. 510, no. 1, pp. 108–121, 2020.

[33] Y. Li, S. Wang and T. Nguyen, "Fault localization with code coverage representation learning," in *IEEE/ACM 43rd Int. Conf. on Software Engineering*, Madrid, Spain, pp. 661–673, 2021.

[34] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis and D. Nickovic, "CPSDebug: Automatic failure explanation in cps models," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1, pp. 69–86, 2021.

[35] R. Assi, W. Masri and C. Trad, "How detrimental is coincidental correctness to coverage-based fault detection and localization? An empirical study," *Software Testing, Verification and Reliability*, vol. 31, no. 5, pp. 1–26, 2021.

[36] M. Wardat, W. Le and H. Rajan, "Deeplocalize: Fault localization for deep neural networks," in *IEEE/ACM 43rd Int. Conf. on Software Engineering (ICSE)*, Madrid, Spain, pp. 251–262, 2021.

[37] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu *et al.,* "A study of effectiveness of deep learning in locating real faults," *Information and Software Technology*, vol. 131, no. 1, pp. 1–16, 2021.

[38] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco *et al.,* "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 3, pp. 35–45, 2007.

[39] M. S. Yang, C. Y. Lai and C. Y. Lin, "A robust EM clustering algorithm for Gaussian mixture models," *Pattern Recognition*, vol. 45, no. 11, pp. 3950–3961, 2012.

[40] X. Xie, F. C. Kuo, T. Y. Chen, S. Yoo and M. Harman, "Provably optimal and human-competitive results in SBSE for spectrum based fault localisation," in *Int. Sym. on Search Based Software Engineering*, St. Petersburg, Russia, pp. 224–238, 2013.