The Journal of Systems and Software 208 (2024) 111900

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Trace matrix optimization for fault localization☆

Jian Hu

School of Big Data and Software Engineering, Chongqing University, Chongqing, China

ARTICLE INFO

Keywords:
Fault localization
Coincidental correctness
Imbalanced data
Data augmentation

ABSTRACT

Fault localization (FL) techniques gather trace information as input data and analyze it to identify the relationship between program statements and failures. Therefore, the input trace matrix is essential for fault localization. However, the current trace matrix faces two main challenges. Firstly, the occurrences of coincidental correctness (CC), which refer to the execution of faulty statements that lead to correct program output, adversely impact the effectiveness of FL. Secondly, the significant disparity in the number of failing and passing test cases poses a data imbalance problem for fault localization. To overcome these issues, we propose TRAIN: a **T**wo-stage t**R**ace m**A**trix opt**I**mizatio**N** method for fault localization. In the first stage of optimization, TRAIN leverages an improved cluster analysis to identify and exclude the CC tests to optimize the trace matrix. Subsequently, in the second stage, TRAIN utilizes data augmentation to enhance the failing test cases to further balance the trace matrix. The optimized trace matrix is then used as input data in the FL pipeline to locate the faulty statements. Through extensive experiments conducted on 330 faulty versions of nine large-sized programs (obtained from Defects4J, ManyBugs, and SIR) using six state-of-the-art FL methods, TRAIN demonstrates remarkable improvements in FL effectiveness.

1. Introduction

During software development and maintenance, debugging is crucial to identify and fix program bugs. However, it is a highly manual and costly process. To reduce these expenses, researchers have developed fault localization techniques that assist in locating faults within the program (Wong et al., 2016; Jones, 2004; Li et al., 2021, 2019; Sohn and Yoo, 2017; Jones and Harrold, 2005). Among these techniques, spectrum-based fault localization (SFL) (Tang et al., 2017) and deep learning-based fault localization (DLFL) (Li et al., 2019) are extensively studied and widely used. Both SFL and DLFL utilize trace matrix data and test results to evaluate each statement's suspiciousness for being faulty, generating a ranked list of statements in descending order of suspiciousness. The concept underlying both techniques is based on the notion that if a statement appears more frequently in failing test cases and less frequently in passing test cases, its suspiciousness value as a potential fault should be higher. Building upon this concept, researchers have developed various SFL techniques (Dstar Wong et al., 2014, Wong Eric Wong et al., 2010, Tarantula Jones and Harrold, 2005, and Ochiai Abreu et al., 2009b) and machine learning models (BP Maru et al., 2019, RBF Wong et al., 2012, MLP Zheng and D. Hu, 2016, CNN Zhang et al., 2019, and RNN Zhang et al., 2021b).

Fig. 1 illustrates the typical workflow of FL. FL executes a test suite and collects raw data in the form of trace matrix (coverage information) and test results (errors). The rows in trace matrix represent the test cases' coverage information and each column represents a statement's coverage information across all test cases in the suite. Regarding the trace matrix, an element x_{ij} equals 1 indicating that the ith test case executes the jth statement; otherwise, x_{ij} equals 0. For the errors, an element e_i equals 1 when the ith test case fails and 0 otherwise. After obtaining the raw data, many FL methods directly use them as input to assess each statement's suspiciousness value and generate a ranked list of program statements in descending order of their suspiciousness values for manual or automated debugging (Wong et al., 2014; Jones and Harrold, 2005; Li et al., 2021; Jones, 2004).

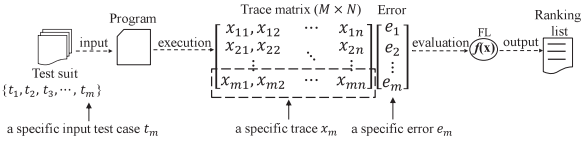

Fig. 1. A typical fault localization process of FL.

Therefore, the raw input trace matrix data is essential for carrying out effective FL. However, the raw data faces two challenges. Firstly, coincidental correctness (CC) (Hierons, 2006; Masri et al., 2009), where a test coincidentally executes a fault but the program produces the correct output, can diminish the FL's effectiveness. Previous studies (Masri

☆ Editor: W. Eric Wong.

E-mail address: jianhu@cqu.edu.cn.

https://doi.org/10.1016/j.jss.2023.111900
Received 30 June 2023; Received in revised form 18 October 2023; Accepted 6 November 2023
Available online 18 November 2023

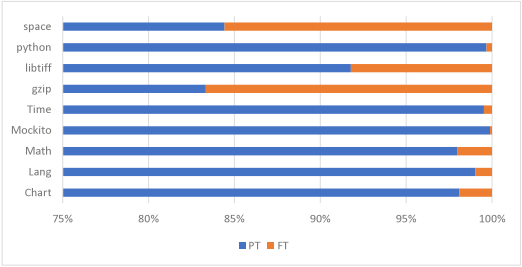

Fig. 2. Statistics of test cases in representative benchmarks.

and Assi, 2010) have assessed the impact of coincidental correctness on fault localization and demonstrated that it undermines the effectiveness of FL. Furthermore, the abundance of coincidental correctness in test suites exacerbates the negative effect on FL effectiveness (Masri and Assi, 2014). Secondly, in practice, the number of failing test cases is typically much smaller than the number of passing test cases, resulting in a class imbalance problem. Fig. 2 showcases the numbers of passing and failing test cases in real benchmarks, where "PT" and "FT" represent the number of passing and failing test cases, respectively. As illustrated in the figure, the number of failing test cases is considerably smaller than that of passing test cases. For instance, consider the *Chart* program from the Defects4J dataset, which has only 92 failing test cases compared to 4707 passing test cases. Prior research (Gong et al., 2012; Zhang et al., 2017) has shown that imbalanced data can significantly reduce the effectiveness of FL techniques, highlighting the significance of balancing input data for these methods.

Therefore, addressing the challenges of coincidental correctness (CC) and data imbalance is crucial in fault localization. Previous studies have indicated that CC tests exhibit similarities to failing tests in terms of executed statements, leading researchers to focus on automatically clustering them (Hierons, 2006; Masri et al., 2009). Consequently, we employ cluster analysis techniques (Feyzi, 2020; Li and Liu, 2012) to identify CC tests as a distinct cluster within a test suite. However, cluster analysis faces limitations when directly applied to high-dimensional raw trace matrices, which are prevalent in real-world benchmarks (Hu et al., 2023). Such matrices often contain significant noise, which can result in inaccurate clustering outcomes and diminish the effectiveness of CC test detection (Xu et al., 2005; Donoho, 2006). To overcome these limitations, we propose an improved cluster analysis technique that combines dimensionality reduction with traditional cluster analysis. This method aims to identify and remove irrelevant statements from the trace matrix. By doing so, we obtain a reduced trace matrix that contains only those statements whose execution influences failure occurrence, thereby facilitating CC detection (Donoho, 2006).

On the other hand, generating a failing test case to achieve balance in the trace matrix is a highly resource-intensive process. This is primarily due to the fact that faulty statements represent only a small fraction of the total statements, and random input tests typically fail to cover these specific faulty statements adequately (Hu et al., 2023). To address this challenge, researchers have developed re-sampling (Gao et al., 2013; Zhang et al., 2017) and under-sampling (Wang et al., 2020) techniques to balance the input data for fault localization. These methods have proven effective in enhancing the features of minority classes. However, traditional data augmentation algorithms have certain limitations. Firstly, they often overlook the distribution of minority data in their algorithms, failing to consider important information from borderline minority data. In reality, the classification of borderline minority data holds greater significance since it is more likely to be misclassified compared to cases further away from the borderline. Secondly, cloning existing data lacks variability in failing tests, potentially leading to over-fitting issues. To overcome these limitations, we employ a borderline data augmentation method (Han et al., 2005). Firstly, instead of randomly selecting from all the minority data, we specifically identify borderline minority data as candidates for data augmentation. This approach ensures that crucial information from these borderline cases is taken into account. Secondly, rather than simply cloning existing data, we synthesize new data to create a more generalized decision region for the minority class, thereby avoiding over-fitting problems and enhancing the variability of failing tests (Chawla et al., 2002).

Building on the aforementioned concept, we propose *TRAIN*: a **T**wo-stage t**R**ace m**A**trix opt**I**mizatio**N** method for fault localization. *TRAIN* encompasses two key stages. In the first stage, *TRAIN* employs improved cluster analysis to identify CC and reverses their test results to initially optimize the input trace matrix. In the second stage, *TRAIN* augments borderline failing test cases to balance the trace matrix to further optimize the input trace matrix. After the two-stage optimization, the trace matrix is utilized as input in FL methods to locate faults in the programs.

To evaluate the effectiveness of *TRAIN*, we use it to enhance six state-of-the-art FL methods (e.g., Dstar Wong et al., 2014, Ochiai Abreu et al., 2009b, Barinel Abreu et al., 2009a), MLP-FL (Zheng and D. Hu, 2016), CNN-FL (Zhang et al., 2019), and RNN-FL (Zhang et al., 2021b). Extensive experiments are conducted on a dataset consisting of 330 faulty versions from nine benchmarks. Furthermore, we compare *TRAIN* with two state-of-the-art optimization methods, re-sampling (Gao et al., 2013) and under-sampling (Wang et al., 2020), on these benchmarks. The findings demonstrate that *TRAIN* significantly enhances the effectiveness of the six FL methods and the two optimization methods.

The contribution of this paper can be summarized as follows.

- We propose a two-stage trace matrix optimization method that effectively addresses the issues of CC and data imbalance in FL, thereby improving the effectiveness of a wide range of FL methods.
- Our study demonstrates the promising potential of combining cluster analysis and data augmentation methods for enhancing the effectiveness of fault localization as a universal data optimization technique.
- Our large-scale experiments comparing *TRAIN* with six state-of-the-art FL methods and two representative data optimization methods showcase the significant improvement in effectiveness achieved by our approach for both original FL and data optimization methods.
- We have made our replication package available online, which includes all relevant code.[1]

The paper follows the following structure: Section 2 provides the necessary background information. In Section 3, a comprehensive explanation of our proposed method is presented. The experimental results and corresponding discussion are presented in Sections 4 and 5, respectively. Section 6 outlines the potential threats to validity that need to be considered. Section 7 discusses the related works in the field. Finally, in Section 8, we conclude the paper summarizing the main findings and contributions of this research.

2. Background

2.1. Fault localization

Spectrum-based Fault Localization (SFL) (Wong et al., 2014) utilizes trace information and test results to identify faulty statements by calculating the suspiciousness of each statement. SFL employs an information model, called the trace matrix, as input data to evaluate the

[1] https://github.com/HuJGithub/TRAIN

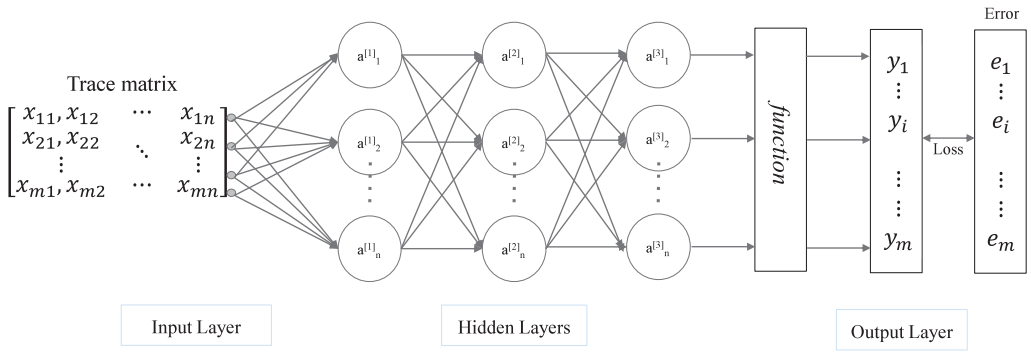

Fig. 3. The typical architecture of DLFL.

Table 1
Typical formulas of SFL.

Name	Formula
Dstar	$Susp(s_j) = \frac{a_{ef}^*}{a_{nf}+a_{ep}}$
Ochiai	$Susp(s_j) = \frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})\times(a_{ef}+a_{ep})}}$
Barinel	$Susp(s_j) = 1 - \frac{a_{ep}}{a_{ep}+a_{ef}}$
Optimal_P	$Susp(s_j) = a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$
Russel_Rao	$Susp(s_j) = \frac{a_{ef}}{a_{nf}+a_{np}+a_{ep}+a_{ef}}$
GP02	$Susp(s_j) = 2(a_{ef} + \sqrt{a_{np}}) + a_{ep}$
GP03	$Susp(s_j) = \sqrt{\lvert a_{ef}^2 - \sqrt{a_{ep}}\rvert}$
GP19	$Susp(s_j) = a_{ef}\sqrt{\lvert a_{ep} - a_{ef} + a_{nf} - a_{np}\rvert}$

statement's suspiciousness. The trace matrix records the runtime information of a test suite. Researchers have proposed numerous SFL techniques such as Dstar (Wong et al., 2014), Ochiai (Abreu et al., 2009b), Barinel (Abreu et al., 2009a), Tarantula (Jones and Harrold, 2005), Jaccard (Chen et al., 2002), and GP02 (Tse, 2014). The equations in Table 1 illustrate some of the typical formulas used for evaluating the suspiciousness in SFL:

The variables used in the equations are defined as follows: a_{ef} represents the number of times a statement is executed in failing test cases, while a_{ep} denotes the number of times a statement is executed in passing test cases. Likewise, a_{nf} and a_{np} signify the number of times a statement is not executed in failing and passing test cases, respectively.

Deep Learning based Fault Localization (DLFL) uses neural networks to develop a fault localization model that assesses the suspiciousness of a statement's potential fault (Guo et al., 2017). The typical architecture of DLFL, depicted in Fig. 3, comprises an input layer, hidden layers, and an output layer.

The deep neural network starts by taking the trace matrix and test result vector as its input, where each element of the matrix represents an input feature. The hidden layers perform mathematical operations on the previous layer inputs, with each neuron calculating a weighted sum of its inputs, applying an activation function, and passing the result to the next layer. By using non-linear activation functions in the hidden layers, the model can learn complex relationships between inputs and outputs. At the output layer, the final prediction or classification is produced based on the computations performed by the hidden layers.

To limit the output to a range between 0 and 1, the model uses the *sigmoid* function (Zhang et al., 1996). A loss function is utilized to measure the training data model's performance by quantifying the difference between the predicted output suspiciousness and the input test result vector. Backpropagation updates the model's weights and biases, propagating the error back through the layers and adjusting the parameters to minimize the error. Optimization algorithms such as Stochastic Gradient Descent (Ruder, 2017), Adam (Kingma and Ba, 2017), and RMSprop (Radford et al., 2016) exist to determine optimal values for the model's weights and biases that minimize the loss function. After training, the model can predict outcomes on new data by feeding virtual tests into the neural network. These tests enable the model to evaluate each statement's suspiciousness and make predictions accordingly.

2.2. Coincidental correctness

Even though current FL methods, including SFL and DLFL, produce promising results, their effectiveness in real-life scenarios may be negatively impacted by coincidental correctness (CC) (Hierons, 2006; Masri et al., 2009). A CC test case refers to a test case that executes a fault but produces a correct program output. Studies (Hierons, 2006; Masri et al., 2009; Masri and Assi, 2010) have indicated that CC is widespread and detrimental to fault localization. To demonstrate that the presence of CC tests results in the underestimation of the faulty statement's suspiciousness, we present an example using SFL dstar (Wong et al., 2014) to calculate the suspiciousness of each statement. For a given statement s_j, its suspiciousness can be computed using Eq. (1).

$$Dstar: Susp(s_j) = \frac{a_{ef}^*}{a_{nf}+a_{ep}} \tag{1}$$

Suppose there are n CC tests that execute the faulty statement s_j with the correct output values under these tests. In this case, the computed suspiciousness value $Susp(s_j)$ of s_j is misleading. If we identified all the CC tests in this example, a_{ef} should be $a_{ef} = a_{ef} + n$ and $a_{ep} = a_{ep} - n$. The updated calculation would be as follows, as expressed in Eq. (2).

$$Dstar: Susp(s_j)' = \frac{(a_{ef}+n)^*}{a_{nf}+a_{ep}-n} \tag{2}$$

It is evident that $Susp(s_j)' \geq Susp(s_j)$, indicating that omitting CC tests would underestimate the suspiciousness of faulty statement s_j.

We also provide a concrete example in Fig. 4 illustrating a program containing a fault located at the third statement across eight test cases. Of these test cases, t1, t2, t3, and t4 are CC test cases as all execute the faulty statement but produce correct program outputs. To demonstrate how CC reduces FL effectiveness, we use GP02, a representative FL method, to compute the suspiciousness of each statement using the execution trace matrix of the example program. With CC, the suspiciousness ranking list is [(6, 12.2), (8, 12.0), (9, 12.0), (1, 10.0), (2, 10.0), (**3, 10.0**), (4, 10.0), (5, 4.2), (7, 4.2)] (the first number is the statement line and the second number is its corresponding suspiciousness value), where the faulty statement ranks at 6th place. Without CC, the suspiciousness ranking list is [(1, 16.0), (2, 16.0), (**3, 16.0**), (4, 16.0), (6, 12.0), (8, 8.0), (9, 8.0), (5, 4.0), (7, 4.0)], and the faulty statement ranks at 3rd place. Therefore, this example illustrates that CC test cases negatively impact FL effectiveness.

input: t1=(2,2), t2=(3,1), t3=(-8,-8), t4=(6,4) t5=(4,5), t6=(3,6), t7=(2,7), t8=(-3,9)

S#		t1	t2	t3	t4	t5	t6	t7	t8
	double P(int a, int b){								
1	int sum=a+b;	•	•	•	•	•	•	•	•
2	int dif=a-b;	•	•	•	•	•	•	•	•
3	int prod=a+b; //correct: prod=a*b	•	•	•	•	•	•	•	•
4	if (a>b)	•	•	•	•	•	•	•	•
5	return dif;		•		•				
6	if (a==b)	•		•		•	•	•	•
7	return sum;	•		•					
8	if (a<b)					•	•	•	•
9	return prod;}					•	•	•	•
	test results	P	P	P	P	F	F	F	F

Fig. 4. An example program with one fault and a set of eight test cases.

2.3. Data augmentation

Previous studies have demonstrated that addressing class imbalance in raw data can be achieved through the utilization of data augmentation techniques (Gao et al., 2013; Zhang et al., 2017; Hu et al., 2023; Zhang et al., 2021a). Basic data augmentation techniques such as flip, rotation, scale, crop, and translation have been widely employed (Krizhevsky et al., 2017; Ciregan et al., 2012). In recent years, re-sampling and under-sampling methods have also been utilized for data augmentation (Gao et al., 2013; Wang et al., 2020). Re-sampling involves duplicating examples of the minority class to augment it, while under-sampling randomly removes some examples from the majority class. However, random re-sampling may lead to over-fitting by making the decision regions of the learner smaller and more specific. Random under-sampling can result in the loss of useful information in the data.

To enhance prediction performance, most classification algorithms aim to accurately learn the boundaries between each class during training. Samples that fall on or near the class boundary are more likely to be misclassified than those further away, making them crucial for classification (Han et al., 2005). Hence, it is crucial to focus on enhancing the data in the class boundary. The minority data can be classified into three distinct sets: the NOISE set, the DANGER set, and the SAFE set, based on the number of neighbors (k). The NOISE set comprises data with no neighboring minority data within the parameter k. The DANGER set comprises data with fewer neighboring minority data than majority data within the parameter k. The SAFE set consists of data with more neighboring minority data than majority data within the parameter k. The data in DANGER set have more majority data than minority data, which are considered to be easily misclassified. Hence, we augment these data to balance the trace matrix for FL.

In Fig. 5, the data distribution is visualized, where red circles represent majority samples and blue circles indicate minority samples. Firstly, we identify the set to which the original minority samples belong (e.g., circle A belongs to NOISE, circle B belongs to DANGER, and circle C belongs to SAFE). Then, new synthetic samples (e.g., circle D) are generated specifically from the borderline examples of the minority class within the DANGER set. The data augmentation method differs from existing data re-sampling methods as it exclusively focuses on over-sampling or reinforcing the borderline minority data within the DANGER set. These samples are more informative and critical for addressing class imbalance.

3. Methodology

3.1. Overview

In this problem formulation, we consider a faulty program denoted as P, which contains N statements. This program is executed by M test cases, denoted as T, where at least one failing test case is present. By running all the test cases, we obtain an execution trace matrix X of size $M \times N$. Each element x_{ij} in the matrix indicates whether statement s_j occurs in test case t_i, with a value of 1, or not, with a value of 0. The error vector e represents the test results, with each element e_i equal to 1 if test case t_i failed, and 0 otherwise. Since the original FL trace matrix serves as a universal input for most FL techniques, *TRAIN* will keep its structure to easily applied alongside a variety of FL techniques.

Our method is a two-stage data optimization process: CC detection and data augmentation, as illustrated in Fig. 6. The first data optimization stage is CC detection, where we identify the CC tests from the trace matrix X and reverse the error vector, changing the value from 0 (passing) to 1 (failing) for these tests. The second data optimization stage is data augmentation. We first identify the borderline minority data that have more nearest neighbors from the majority class than the minority class, categorizing them as danger minority data. And then we proceed to augment the danger minority data until the test suite is balanced, meaning the number of failing test cases equals the number of passing test cases.

The two-stage data optimization process is described in detail below.

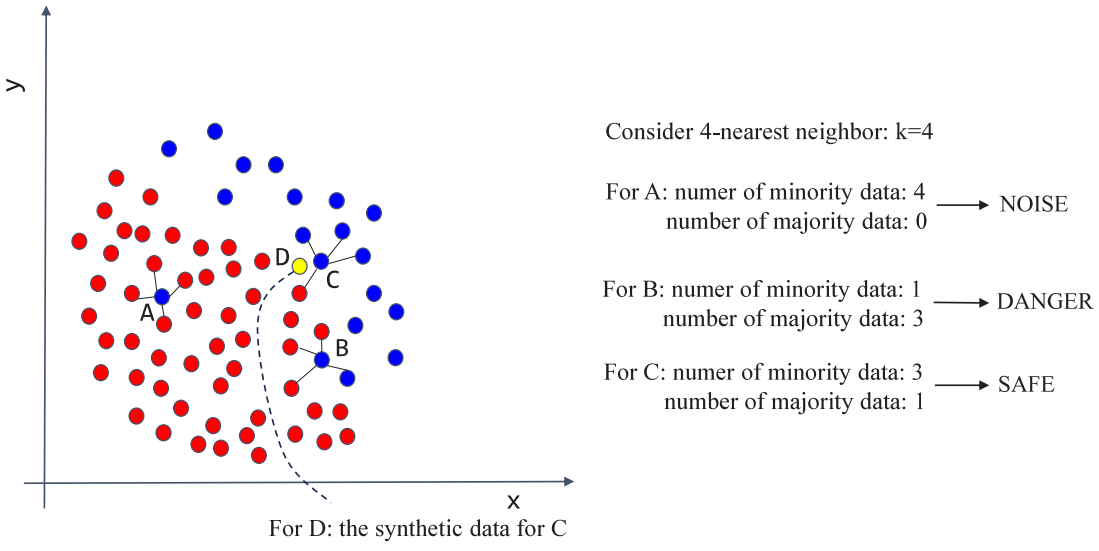

Fig. 5. The distribution of data set. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

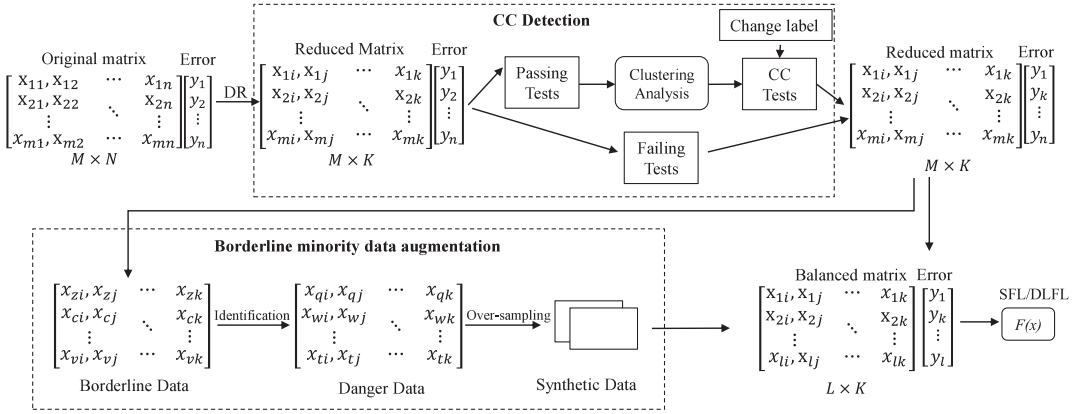

Fig. 6. The overview architecture of *TRAIN*.

3.2. Data optimization stage 1: Coincidental correctness detection

Previous studies (Hierons, 2006; Masri et al., 2009) have proposed that CC tests exhibit similarities to failing tests in terms of the executed statements and have attempted to automatically cluster them together. In our method, we utilize a cluster analysis to partition all passing tests into two clusters: true passing tests and CC tests. However, it is important to note that the trace matrix for most real programs tends to be high-dimensional, where the programs are large in size but only several lines of codes or even one statement is responsible for program failure (Xie et al., 2022). High-dimensional data introduces a significant amount of noise, which can lead to reduced effectiveness of cluster analysis methods for CC detection (Donoho, 2006; Xu et al., 2005; Li and Orso, 2020; Zhang et al., 2005).

To tackle this challenge, we propose an enhanced cluster analysis method that discerns varying degrees of definiteness linked to failing and passing test cases. Failing tests provide definite information that the executed statements by them must contain the faulty statements. Conversely, unexecuted statements by failing tests cannot be considered faulty under the current test suite. Previous research has shown that minimizing the reliance on unreliable information for unexecuted statements is beneficial for fault localization (Xie et al., 2010). Therefore, by focusing solely on the statements executed by failing test cases, we can filter out fault irrelevant statements and reduce the dimensionality of the input data. This reduction in dimensionality helps mitigate the impact of noise and improve the effectiveness of our approach in detecting CC tests accurately.

Algorithm 1 describes the CC detection process to optimize the raw trace matrix. This algorithm takes the raw trace matrix X and error vector e as inputs and returns the error vector e_{wos} with corrected labels. It begins by identifying the failing test cases (line 1) and then determines the columns of the trace matrix that were not executed by these failing tests (line 2). The algorithm reduces the dimensionality of the trace matrix by removing the unexecuted columns (lines 3–5). Next, it randomly selects a failing test case as the initial centroid vector for the CC (line 6). The algorithm calculates the Euclidean distance between all passing test cases and this initial centroid, identifying the test case with the largest distance as the centroid for true passing test cases (lines 7–13). The algorithm iteratively updates the centroids for CC and true passing test cases until convergence (lines 14–29). In each iteration, it calculates the Euclidean distance of each passing test case with the current centroids and assigns them to the set with the smaller distance (lines 16–25). The algorithm then updates the centroids based on the newly assigned sets (lines 26–28). Once convergence is achieved, the corresponding error labels are changed from 0 to 1 according to the CC tests (lines 30–31). Finally, the algorithm returns the error vector e_{wos} with corrected labels (line 33).

Algorithm 1 CC detection.

Input:
X: coverage matrix with the size of $M \times N$
e: error vector
Output:
e_{wos}: error vector without CC tests

1: *failTests* = failing test cases
2: *notExeindex* = uncovered columns by any failing test
3: **for** col in $notExeindex$ **do**
4: $X.remove(X[col])$
5: **end for**
6: $V_{cc} = random(failTests)$: initial centroid vector for CC
7: $passTests = X.remove(failTests)$
8: **for** each pt in $passTests$ **do**
9: distance.add($euclidDis(pt, V_{cc})$)
10: **end for**
11: $maxDis = max(distance)$
12: $candidates = X[indexof(maxDis)]$
13: $V_{tp} = random(candidates)$
14: $V_{cc_0} = V_{tp_0} = none$
15: **while** $V_{cc_0} \neq V_{cc}$ and $V_{tp_0} \neq V_{tp}$ **do**
16: $T_{cc} = T_{tp} = \varnothing$
17: **for** each x in $passTests$ **do**
18: $dis_{cc} = euclidDis(x, V_{cc})$
19: $dis_{tp} = euclidDis(x, V_{tp})$
20: **if** $dis_{cc} < dis_{tp}$ **then**
21: $T_{cc}.add(x)$
22: **else**
23: $T_{tp}.add(x)$
24: **end if**
25: **end for**
26: $V_{cc_0}, V_{tp_0} = V_{cc}, V_{tp}$
27: V_{cc} = calculate the centroid of T_{cc}
28: V_{tp} = calculate the centroid of T_{tp}
29: **end while**
30: **for** each cc in T_{cc} **do**
31: $e_{woc} = e.change(index(cc))$
32: **end for**
33: **return** e_{wos}

3.3. Data optimization stage 2: Data augmentation

After the CC detection stage, we obtain an optimized trace matrix with a shape of $M \times N$. This optimized matrix no longer contains CC tests but may still be imbalanced. In the second optimization stage of *TRAIN*, we leverage an borderline over-sampling technique (Han et al.,

2005) to balance the input trace matrix and further optimize the data. The augmentation method differs from existing over-sampling methods in FL, where either all minority examples or a random subset of the minority class is over-sampled (Gao et al., 2013; Zhang et al., 2021a, 2017). Instead, in *TRAIN*, we focus on over-sampling the minority samples near the decision boundary. This is because data on the borderline are more likely to be misclassified than those that are far from the boundary, making them more important for classification purposes.

Algorithm 2 outlines the procedure for optimizing the trace matrix by over-sampling the data. The algorithm first identifies the borderline data and then employs the Synthetic Minority Over-sampling Technique (SMOTE) as the data augmentation method. To avoid over-fitting, we choose SMOTE over the alternative of cloning data, as it synthesizes more generalized data than cloning the existing minority data (Chawla et al., 2002). The inputs for the algorithm consist of the previously optimized trace matrix X, the error vector e_{wos} with corrected labels, and the number of nearest neighbors k. It outputs synthetic failing test cases denoted as $X_{synthetic}$.

To begin, the algorithm separates the majority and minority data from the input trace matrix X (line 1). Subsequently, it determines the required number of failing test cases to generate (line 2). The next step involves initializing an empty set called $dangerSet$, followed by adding failing test cases with a higher count of majority samples that are closest to them into the danger set (lines 3–9). After identifying the danger data, the algorithm creates an empty set called $X_{synthetic}$ to store the synthetic failing test cases (line 10). It then proceeds to select a borderline failing test case from the danger set $dangerSet$ and identifies its nearest failing test case (lines 12–18). The difference between the selected borderline failing test case and its nearest failing test case is calculated (line 19). This difference is then multiplied by a random number between 0 and 1 and added to the borderline failing test case under consideration (lines 20–21). And then a synthetic failing test case is generated at a random point along the line segment connecting the borderline test case and its nearest failing test case. Next, the algorithm appends this newly generated synthetic failing test case $t_{synthetic}$ to the set of synthetic failing test cases $X_{synthetic}$ (line 23). This generation process continues until the number of passing test cases matches the number of failing test cases. Finally, the algorithm returns the resulting set of synthetic failing test cases $X_{synthetic}$ (line 24).

Lastly, *TRAIN* utilizes both the original dataset X and the augmented data set $X_{synthetic}$ for subsequent FL techniques such as SFL and DLFL to address the issue of imbalanced data in order to mitigate potential bias.

Algorithm 2 Data augmentation using borderline data over-sampling

Input:
X: previous optimized trace matrix with the size of $M \times K$
e_{wos}: error vector with corrected labels
k: number of nearest neighbors
Output:
$X_{synthtic}$: synthetic failing test cases

1: X_f, X_p = Find the majority tests and minority tests in e_{wos}
2: $N = len(X_f) - len(X_p)$: calculate the number of failing test cases need to generate
3: initialize the danger set $dangerSet$ to $\varnothing$
4: **for** each failing test *failTest* in X_f **do**
5: nearSet = find the nearest k samples
6: **if** passing test cases outnumber the failing test cases in nearSet **then**
7: $dangerSet$.add(*failTest*)
8: **end if**
9: **end for**
10: Initialize synthetic failing test case set $X_{synthtic}$ to $\varnothing$
11: **for** $i = 1; i \leq N; i++$ **do**
12: *fail* = select a failing test in $dangerSet$
13: Initialize distance list $distance$ to $\varnothing$
14: **for** $i = 1; i \leq len(X_f); i++$ **do**
15: $distance[i] = euclidDis(fail, X_f[i])$
16: **end for**
17: $minDis = selectMin(distance)$
18: $nearSample = X[indexof(minDis)]$
19: dif = *nearSample* - $fail$
20: γ = random number between 0 and 1
21: $t_{synthtic} = fail + \gamma \times dif$
22: $X_{synthtic}.append(t_{synthtic})$
23: **end for**
24: **return** $X_{synthtic}$

4. Experiment

4.1. Experimental setup

Baselines We employed six state-of-the-art FL methods as our baselines. For SFL baselines, we used Dstar (Wong et al., 2014), Ochiai (Abreu et al., 2009b), and Barinel (Abreu et al., 2009a). These formulas were selected based on previous research that identified them as optimal choices through theoretical and empirical analyses (Tse, 2014; Xie et al., 2013). As for DLFL baselines, we chose MLP-FL (Zheng and D. Hu, 2016), CNN-FL (Zhang et al., 2019), and RNN-FL (Zhang et al., 2021b). Our selection was motivated by two factors. Firstly, our study focuses on statement-level FL, whereas recent DLFL baselines such as DeepFL (Li et al., 2019) and FLUCCS (Sohn and Yoo, 2017) target method-level FL. Secondly, our approach solely relies on raw test case data, without incorporating complex source code structures like AST, which can pose challenges when dealing with larger projects. Consequently, we excluded other baselines, such as DEEPRL4FL (Li et al., 2021), which require these structures for data augmentation.

Benchmarks Table 2 provides a summary of the nine subject programs used in this study. It includes the program names ("Programs"), a brief description of their functionality ("Description"), the number of faulty versions ("Versions"), the size in thousands of lines of code ("KLOC"), the number of test cases ("Tests") and the type of faults ("Type"). The selected programs consist of Chart, Math, Lang, Time, and Mockito from the Defects4J dataset,[2] gzip, libtiff, and python from ManyBugs,[3] and Space from the SIR dataset.[4] The subject programs were chosen for three specific reasons. Firstly, these programs are widely used and considered large-sized in fault localization research (e.g., Jones, 2004; Li et al., 2021; Sohn and Yoo, 2017; Wong et al., 2014; Abreu et al., 2009b; Jones and Harrold, 2005; Abreu et al., 2009a; Zheng and D. Hu, 2016; Zhang et al., 2019, 2021b). Secondly, they were easily accessible, ensuring comparability and reproducibility of the studies. Lastly, all the selected programs contain real faults, allowing us to evaluate our performance in real-world scenarios. ***Environment*** The experiment was conducted on a Linux server running a 64-bit architecture with 16 Intel(R) Xeon CPUs and 128 GB of RAM. The operating system used was Ubuntu 16.04.3. To run the experiment, it is necessary to have Python packages of specific versions installed: pandas == 0.25.1, chardet == 3.0.4, numpy == 1.16.5, and torch == 1.9.0.

4.2. Evaluation metrics

We employed five widely-accepted metrics to assess the performance of *TRAIN* in fault localization. These metrics include Top-K accuracy (Kochhar et al., 2016), Mean Average Rank (Li and Zhang, 2017), Mean First Rank (Li and Zhang, 2017), Relative Improvement (Debroy et al., 2010), and Wilcoxon signed-rank tests (Wilcoxon, 1944).

Top-K accuracy: This metric measures the ability of a FL method to correctly rank faulty versions within the top K positions. We evaluated

[2] https://github.com/rjust/defects4j
[3] https://repairbenchmarks.cs.umass.edu/
[4] https://sir.csc.ncsu.edu/portal/index.php

Table 2
Subject programs.

Programs	Description	Versions	KLOC	Tests	Type
Chart	Java chart library	26	96	2205	Real
Math	Apache commons-math	106	85	3602	Real
Lang	Apache commons-lang	65	22	2245	Real
Time	Standard date and time library	27	28	4130	Real
Mokito	Mocking framework for Java	38	67	1075	Real
gzip	Data compression	5	491	12	Real
libtif	Image processing	12	77	78	Real
python	General-purpose language	8	10	355	Real
space	ADL interpreter	38	6	13 585	Real
Average		45.8	97.2	3521.4	

the performance using K values of 1, 5, and 10 based on previous studies (Heris and Keyvanpour, 2019; Küçük et al., 2021). A higher value of Top-K indicates better fault localization effectiveness.

Mean Average Rank (MAR): For each faulty version, we calculated the average rank of all its faulty statements in the ranking list. The Mean Average Rank is then computed as the mean of these average ranks across all versions.

Mean First Rank (MFR): This metric computes the rank of the first identified faulty statement for each version and calculates the average rank across all versions.

Relative Improvement (RImp): RImp is used to compare the number of statements examined by different fault localization methods. It quantifies the ratio of statements that need to be examined using our method compared to other methods. A lower Relative Improvement indicates better performance.

Wilcoxon signed-rank tests (WSR): WSR test is a non-parametric statistical hypothesis test used to evaluate the differences between pairs of measurements, which evaluates the statistical significance of our results.

It is important to mention that when multiple statements have the same suspiciousness value, we implemented a statement order-based strategy (Xu et al., 2011). This strategy involved sorting them based on their line numbers in ascending order during the evaluation process.

4.3. Research questions and results

4.3.1. RQ1. How does TRAIN perform in localizing real faults compared with original state-of-the-art SFL techniques?

Top-K Accuracy, MAR, and MFR. We evaluated the effectiveness of three state-of-the-art SFL methods (Dstar, Ochiai, and Barinel) under two scenarios: original and *TRAIN*. According to Parnin's study (Parnin and Orso, 2011a), the rank of the faulty statement is a reliable metric for evaluating fault localization techniques. Therefore, we conducted a performance comparison between *TRAIN* and SFL baselines using the Top-K, MAR, and MFR metrics. In line with previous studies (Heris and Keyvanpour, 2019; Küçük et al., 2021; Jones, 2004), we adopted values of K = 1, 5, and 10 for the Top-K metric. Table 3 presents the experimental results, with better performance indicated in bold for readability. The results reveal that *TRAIN* outperforms the baselines. Taking the Mokito data as an example, *TRAIN* shows improvements of 50.00%, 16.67%, and 23.08% respectively when evaluated with the Top-1, Top-5, and Top-10 metrics compared to the Dstar baseline. These results demonstrate that *TRAIN* effectively enhances the fault localization accuracy of the baseline in the Top-K evaluation.

Moreover, the MFR and MAR metrics demonstrate that *TRAIN* consistently achieves lower ranks compared to the baselines across all SFL techniques. To provide a visual representation of this, Fig. 7 illustrates the distribution of MFR and MAR for SFL baselines and *TRAIN*. The results clearly indicate that *TRAIN* consistently identifies at least one faulty line earlier and efficiently identifies all faulty lines. As an example, when considering Ochiai, the MAR of *TRAIN* is 370.40, significantly smaller than the MAR of 590.19 achieved by the baseline. This substantial difference underscores the effectiveness of our proposed method.

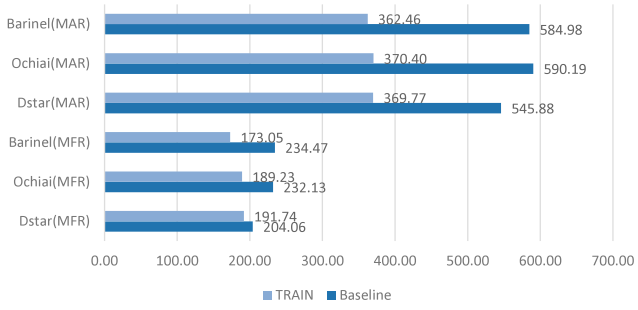

Fig. 7. MFR and MAR values of SFL baseline and *TRAIN*.

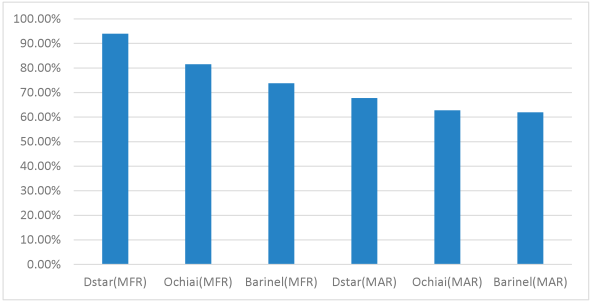

Fig. 8. The RImp values of *TRAIN* over SFL baselines.

RImp distribution. To provide a comprehensive analysis of the effectiveness of *TRAIN*, we utilized the RImp metric to evaluate its performance compared to three SFL baselines without *TRAIN*. Fig. 8 presents the distribution of RImp scores for MFR and MAR, showcasing the impact of *TRAIN* on the localization effectiveness of the SFL baselines. As depicted in Fig. 8, all methods achieved RImp scores below 100%, indicating that *TRAIN* significantly improved the fault localization effectiveness of the three baselines. For instance, the percentage of statements examined ranged from 61.96% in Barinel to 67.74% in Dstar for MAR. By employing *TRAIN*, minimum saving of 32.26% (100%−67.74% = 32.26%) was achieved in Dstar, while maximum saving of 38.04% (100%−61.96% = 38.04%) was observed in Barinel. These results clearly demonstrate that *TRAIN* can reduce the percentage of examined statements by 32.26% to 38.04% across the SFL methods.

Statistical comparison. To investigate the statistical significance of the difference between using *TRAIN* and not using *TRAIN* in the baselines, we employed the Wilcoxon-Signed-Rank (WSR) Test (Wilcoxon, 1944) with a Bonferroni correction (Abdi, 2007). This non-parametric statistical hypothesis test is applicable for comparing pairs of measurements $F(x)$ and $G(y)$. For our experiment, we conducted 30 paired WSR tests using the ranks of faulty statements as the pairs of measurements $F(x)$ and $G(y)$. Each test employed a p-value check at a significance

Table 3
The results of TOP-1, TOP-5, TOP-10, MFR and MAR of original SFL method and *TRAIN*.

program	Scenario	TOP-1	TOP-5	TOP-10	MFR	MAR
Chart	Dstar(baseline)	**2**	**8**	**12**	322.88	828.01
	Dstar(TRAIN)	**2**	**8**	**12**	**226.8**	**564.64**
	Ochiai(baseline)	**2**	9	14	209.24	697.17
	Ochiai(TRAIN)	**2**	**10**	**15**	**176.32**	**567.64**
	Barinel(baseline)	**2**	**9**	**13**	141.04	655.42
	Barinel(TRAIN)	**2**	**9**	**13**	**127.45**	**583.87**
Time	Dstar(baseline)	**2**	8	10	398.63	675.20
	Dstar(TRAIN)	**2**	**11**	**12**	**378.96**	**622.51**
	Ochiai(baseline)	**2**	8	10	598.96	874.50
	Ochiai(TRAIN)	**2**	**10**	**12**	**486.04**	**646.26**
	Barinel(baseline)	**2**	8	10	600.78	875.55
	Barinel(TRAIN)	**2**	**10**	**11**	**541.59**	**579.26**
Lang	Dstar(baseline)	**5**	24	35	29.14	81.61
	Dstar(TRAIN)	**5**	**25**	**36**	**27.38**	**76.90**
	Ochiai(baseline)	**5**	24	35	30.45	60.41
	Ochiai(TRAIN)	**5**	**25**	**37**	**29.82**	**51.91**
	Barinel(baseline)	5	24	35	33.17	61.48
	Barinel(TRAIN)	**6**	**25**	**36**	**29.20**	**61.54**
Math	Dstar(baseline)	15	36	42	59.84	315.93
	Dstar(TRAIN)	**19**	**39**	**48**	**57.92**	**304.34**
	Ochiai(baseline)	14	36	42	61.35	209.81
	Ochiai(TRAIN)	**19**	**39**	**49**	**58.63**	**208.23**
	Barinel(baseline)	14	38	45	68.24	212.46
	Barinel(TRAIN)	**19**	**39**	**49**	**63.46**	**210.93**
Mokito	Dstar(baseline)	4	12	13	254.39	439.27
	Dstar(TRAIN)	**6**	**14**	**16**	**176.67**	**387.36**
	Ochiai(baseline)	3	11	13	201.86	398.89
	Ochiai(TRAIN)	**6**	**13**	**15**	**112.11**	**293.30**
	Barinel(baseline)	3	**11**	13	211.89	402.23
	Barinel(TRAIN)	**4**	**11**	**16**	**114.33**	**295.76**
gzip	Dstar(baseline)	**0**	**1**	**2**	66.80	85.43
	Dstar(TRAIN)	**0**	**1**	**2**	**36.80**	**68.35**
	Ochiai(baseline)	**0**	**1**	**2**	103.80	133.93
	Ochiai(TRAIN)	**0**	**1**	**2**	**37.60**	**107.15**
	Barinel(baseline)	**0**	**1**	**2**	103.80	132.88
	Barinel(TRAIN)	**0**	**1**	**2**	**38.00**	**106.30**
libtif	Dstar(baseline)	1	2	3	98.60	1158.83
	Dstar(TRAIN)	**1**	**2**	**3**	**78.40**	**556.63**
	Ochiai(baseline)	**1**	**2**	**3**	98.80	1266.96
	Ochiai(TRAIN)	**1**	**2**	**3**	**78.60**	**546.76**
	Barinel(baseline)	**1**	**2**	**3**	106.40	1274.62
	Barinel(TRAIN)	**1**	**2**	**3**	**78.40**	**554.70**
python	Dstar(baseline)	**0**	**0**	**0**	229.21	824.76
	Dstar(TRAIN)	**0**	**0**	**0**	**166.50**	**699.76**
	Ochiai(baseline)	**0**	**0**	**0**	291.04	1065.15
	Ochiai(TRAIN)	**0**	**0**	**0**	**142.75**	**765.15**
	Barinel(baseline)	**0**	**0**	**0**	293.89	1065.15
	Barinel(TRAIN)	**0**	**0**	**0**	**143.25**	**782.68**
space	Dstar(baseline)	**3**	5	10	377.09	503.88
	Dstar(TRAIN)	**3**	**6**	**13**	**376.18**	**447.50**
	Ochiai(baseline)	2	5	10	493.71	604.85
	Ochiai(TRAIN)	**3**	**6**	**13**	**381.24**	**537.17**
	Barinel(baseline)	**2**	**3**	8	551.06	585.06
	Barinel(TRAIN)	**2**	**3**	**9**	**438.59**	**519.60**
total	Dstar(baseline)	32	96	127	204.06	545.88
	Dstar(TRAIN)	**38**	**106**	**142**	**191.74**	**369.77**
	Ochiai(baseline)	29	96	129	232.13	590.19
	Ochiai(TRAIN)	**38**	**106**	**146**	**189.23**	**370.40**
	Barinel(baseline)	29	96	129	234.47	584.98
	Barinel(TRAIN)	**36**	**100**	**139**	**173.05**	**362.46**

level of 0.05. Specifically, we utilized the list of ranks of faulty statements when using *TRAIN* across all faulty versions of all programs as

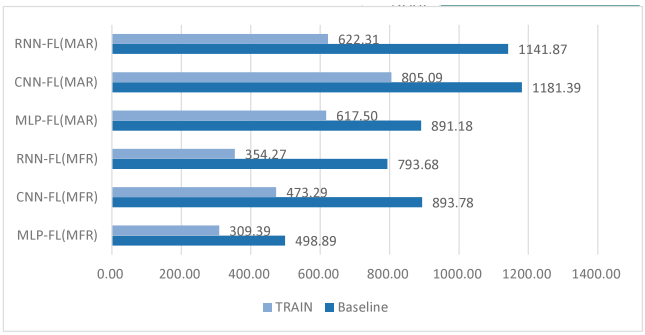

Fig. 9. MFR and MAR values of DLFL baseline and *TRAIN*.

the measurements of F(x). In contrast, the measurements of $G(y)$ were obtained from the list of ranks of faulty statements without using *TRAIN* in all faulty versions of all programs. In the analysis, if the calculated p-value is less than 0.05, we accept the alternative hypothesis (denoted as $H1$), otherwise, we accept the null hypothesis (denoted as $H0$).

The outcomes of the WSR for MFR and MAR are presented in Table 4, which displays the corresponding p-values for each test. Analyzing these results, we can conclude that the ranks of faulty statements achieved using *TRAIN* across all three SFL methods are significantly smaller than those obtained from the baselines. This finding demonstrates the superior performance of *TRAIN* across all cases. For example, when evaluating Dstar for the Math program, the p-values for the right-tailed, left-tailed, and two-tailed tests are 1.000, 6.19E−21, and 2.51E−29, respectively. Accepting the null hypothesis implies that *TRAIN* exhibits a significant improvement over the original SFL method, indicating that it requires fewer lines of code to be examined in comparison. From a statistical standpoint, we can conclude that *TRAIN* outperforms the three state-of-the-art SFL methods.

Summary for RQ1 *In RQ1, we examined the effectiveness of TRAIN in comparison to the original SFL baselines. Based on our findings and the results of statistical analysis, it can be asserted that TRAIN has outperformed the original SFL baselines. This outcome highlights the potential of CC detection and data augmentation in enhancing the effectiveness of SFL methods.*

4.3.2. RQ2. How does TRAIN perform in localizing real faults compared with original state-of-the-art DLFL techniques?

Top-K Accuracy, MAR and MFR We assessed the effectiveness of *TRAIN* on DLFL methods by comparing it with three cutting-edge techniques, namely MLP-FL, CNN-FL, and RNN-FL. We also evaluated their performance using five metrics: Top-1, Top-5, Top-10, MAR, and MFR. The experimental results, displayed in Table 5, indicate that *TRAIN* outperformed the DLFL baselines for all programs. Specifically, when compared to the original CNN-FL, we observed substantial enhancements of 300%, 300%, and 250% for the Top-1, Top-5, and Top-10 metrics, respectively.

Additionally, Fig. 9 presents the distribution of MFR and MAR for DLFL baselines. It clearly demonstrates that *TRAIN* exhibited superior performance in terms of mean average ranks and mean first ranks across all programs. For instance, in the case of RNN-FL, the MFR achieved by *TRAIN* was 354.27, which is significantly lower than the baseline's MFR of 793.68. This substantial disparity underscores the high effectiveness of our proposed method.

RImp distribution To comprehensively evaluate the effectiveness of *TRAIN*, we employed RImp as a metric. Fig. 10 illustrates the distribution of MFR and MAR RImp scores for *TRAIN* in comparison with three DLFL baselines. As depicted, all methods achieved RImp

Table 4
Statistical comparison for *TRAIN* versus the SFL baselines.

Comparison		One-tailed(right)	One-tailed(left)	Two-tailed	conclusion
	Chart	1.000	7.82E−15	1.13E−14	**better**
	Time	1.000	6.53E−20	1.48E−10	**better**
	Lang	1.000	2.03E−26	2.68E−26	**better**
	Math	1.000	6.19E−21	2.51E−29	**better**
Dstar(TRAIN)	Mokito	1.000	7.25E−16	7.17E−28	**better**
vs	gzip	0.995	8.91E−04	3.37E−05	**better**
Dstar	libtif	0.999	8.38E−10	7.73E−09	**better**
	python	0.967	7.77E−04	1.14E−05	**better**
	space	0.998	5.63E−05	3.94E−07	**better**
	total	1.000	6.14E−19	6.04E−20	**better**
	Chart	1.000	3.82E−24	6.45E−16	**better**
	Time	1.000	7.99E−30	2.83E−11	**better**
	Lang	0.987	1.78E−06	6.56E−05	**better**
	Math	1.000	6.19E−21	2.51E−29	**better**
Ochiai(TRAIN)	Mokito	1.000	3.42E−22	2.9E−25	**better**
vs	gzip	0.982	2.32E−04	5.14E−04	**better**
Ochiai	libtif	1.000	7.4E−19	2.5E−10	**better**
	python	0.996	1.59E−12	4.4E−08	**better**
	space	1.000	5.17E−19	8.5E−10	**better**
	total	1.000	6.73E−22	5.47E−27	**better**
	Chart	1.000	6.64E−28	8.34E−13	**better**
	Time	1.000	8.02E−21	6.14E−18	**better**
	Lang	0.989	1.72E−07	3.3E−06	**better**
	Math	1.000	3.13E−15	4.52E−12	**better**
Barinel(TRAIN)	Mokito	1.000	6.19E−21	2.51E−13	**better**
vs	gzip	1.000	4.16E−19	6.24E−12	**better**
Barinel	libtif	0.995	3.57E−08	5.75E−09	**better**
	python	1.000	7.13E−21	7.86E−23	**better**
	space	0.997	3.99E−10	8.98E−11	**better**
	total	1.000	6.19E−21	2.51E−29	**better**

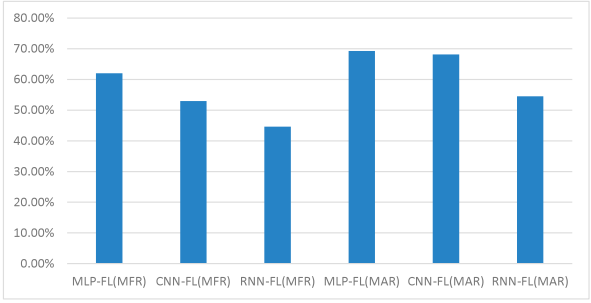

Fig. 10. The RImp values of *TRAIN* over DLFL baselines.

scores below 100%, indicating that *TRAIN* significantly improved the fault localization effectiveness of the three baselines. For instance, when considering MFR, the percentage of statements examined ranged from 44.64% in RNN-FL to 62.02% in MLP-FL. By utilizing *TRAIN*, substantial saving of up to 55.36% (100%−44.64% = 55.36%) was achieved in RNN-FL, while the minimum saving observed was 37.98% (100%−62.02% = 37.98%) in MLP-FL. These results unequivocally demonstrate that *TRAIN* effectively reduced the percentage of examined statements by 37.98% to 55.36% across all DLFL methods.

Statistical comparison The results of the WSR Tests for MFR and MAR between *TRAIN* and the baseline DLFLs are presented in Table 6. These findings reveal that the ranks of faulty statements achieved by *TRAIN* across all three DLFL methods are significantly lower than those attained by the baselines, indicating superior performance in all cases. For instance, when considering MLP-FL for the Chart program, the p-values for the right-tailed, left-tailed, and two-tailed tests are 1.000, 8.47E−8, and 4.0E−16, respectively. Accepting the null hypothesis implies that *TRAIN* is significantly better than the original DLFL method, suggesting that *TRAIN* requires a lower number of lines of code to be examined compared to the original DLFL methods. From a statistical perspective, we can conclude that *TRAIN* achieves better results when compared to the three state-of-the-art DLFL methods.

Summary for RQ2 *In RQ2, we investigated the effectiveness of TRAIN compared to three original DLFL baselines. Our findings, supported by statistical analysis, clearly demonstrate that TRAIN outperforms the original baselines to a significant degree. These results show that CC detection and data augmentation have substantial potential for enhancing the effectiveness of DLFL methods.*

4.3.3. RQ3. How does TRAIN perform in localizing real faults compared with the data re-sampling and under-sampling methods?

In addition to the origin methods, our approach is compared with two data optimization methods: re-sampling method (Gao et al., 2013; Zhang et al., 2017, 2021a) and under-sampling method (Wang et al., 2020). The re-sampling method involves replicating minority samples, while the under-sampling method focuses on removing majority samples. For further information on these methods, refer to Gao et al. (2013) and Wang et al. (2020), respectively.

Top-K Accuracy, MAR and MFR Table 7 presents the results for the Top-K, MAR, and MFR metrics comparing our method with two representative data optimization methods. As demonstrated in Table 7, *TRAIN* consistently outperforms both re-sampling and under-sampling methods in all SFL cases and most DLFL cases. Specifically, when considering CNN-FL as an example, *TRAIN* successfully locates 8, 28, and 42 faults for the Top-1, Top-5, and Top-10 metrics, respectively. These findings reveal improvements of 300%, 75%, and 50% for the Top-1, Top-5, and Top-10 metrics compared to under-sampling. Additionally, compared to re-sampling, *TRAIN* achieves improvements of 60%, 21%, and 20% for the Top-1, Top-5, and Top-10 metrics.

Furthermore, the MFR and the MAR of *TRAIN* are lower than the two representative data optimization methods in all cases. This indicates that *TRAIN* performs better than the other methods. Fig. 11 visually presents the distribution of MFR for re-sampling, under-sampling,

Table 5
The results of TOP-1, TOP-5, TOP-10, MFR and MAR of original DLFL method and *TRAIN*.

Program	Scenario	TOP-1	TOP-5	TOP-10	MFR	MAR
Chart	MLP-FL(baseline)	**1**	2	6	691.84	861.04
	MLP-FL(TRAIN)	**1**	**6**	**8**	**470.96**	**497.35**
	CNN-FL(baseline)	**0**	0	1	617.76	713.99
	CNN-FL(TRAIN)	**0**	**1**	**2**	**495.08**	**534.06**
	RNN-FL(baseline)	0	2	3	598.64	747.80
	RNN-FL(TRAIN)	**1**	**4**	**6**	**529.28**	**561.23**
Time	MLP-FL(baseline)	0	0	0	1359.33	2073.88
	MLP-FL(TRAIN)	**1**	**6**	**9**	**858.81**	**1328.59**
	CNN-FL(baseline)	0	0	0	2513.44	2736.65
	CNN-FL(TRAIN)	**0**	**1**	**1**	**1501.26**	**2023.08**
	RNN-FL(baseline)	0	0	0	1796.22	2326.93
	RNN-FL(TRAIN)	**1**	**4**	**9**	**563.85**	**759.68**
Lang	MLP-FL(baseline)	2	10	16	211.88	304.34
	MLP-FL(TRAIN)	**5**	**24**	**31**	**103.46**	**219.98**
	CNN-FL(baseline)	0	2	6	372.40	412.22
	CNN-FL(TRAIN)	**3**	**10**	**13**	**149.40**	**382.57**
	RNN-FL(baseline)	0	4	10	386.18	466.29
	RNN-FL(TRAIN)	**3**	**16**	**26**	**110.46**	**145.39**
Math	MLP-FL(baseline)	0	5	10	615.04	1026.64
	MLP-FL(TRAIN)	**7**	**17**	**23**	**241.56**	**662.86**
	CNN-FL(baseline)	0	3	3	966.35	1058.76
	CNN-FL(TRAIN)	**2**	**9**	**15**	**242.58**	**814.06**
	RNN-FL(baseline)	1	5	5	1106.98	1373.72
	RNN-FL(TRAIN)	**4**	**18**	**23**	**290.65**	**617.41**
Mokito	MLP-FL(baseline)	0	1	3	502.22	763.85
	MLP-FL(TRAIN)	**3**	**4**	**6**	**256.28**	**349.16**
	CNN-FL(baseline)	**1**	1	1	830.78	1048.27
	CNN-FL(TRAIN)	**1**	**2**	**3**	**464.67**	**460.31**
	RNN-FL(baseline)	0	1	1	651.81	922.51
	RNN-FL(TRAIN)	**0**	**2**	**2**	**487.81**	**333.89**
gzip	MLP-FL(baseline)	0	0	**1**	75.40	185.50
	MLP-FL(TRAIN)	**1**	**1**	**1**	**49.80**	**82.12**
	CNN-FL(baseline)	**0**	**0**	**0**	150.80	177.50
	CNN-FL(TRAIN)	**0**	**0**	**0**	**64.60**	**70.90**
	RNN-FL(baseline)	**0**	**0**	0	173.00	245.28
	RNN-FL(TRAIN)	**0**	**0**	**1**	**65.60**	**60.27**
libtif	MLP-FL(baseline)	0	1	2	281.60	1337.42
	MLP-FL(TRAIN)	**1**	**3**	**3**	**228.40**	**1157.66**
	CNN-FL(baseline)	**1**	**1**	**1**	239.20	1469.42
	CNN-FL(TRAIN)	**1**	**1**	**1**	**201.80**	**1201.58**
	RNN-FL(baseline)	**1**	**1**	**2**	740.60	1994.30
	RNN-FL(TRAIN)	**1**	**1**	**2**	**323.00**	**1726.58**
python	MLP-FL(baseline)	**0**	**0**	**0**	234.88	685.42
	MLP-FL(TRAIN)	**0**	**0**	**0**	**113.50**	**627.37**
	CNN-FL(baseline)	**0**	**0**	**0**	307.44	832.79
	CNN-FL(TRAIN)	**0**	**0**	**0**	**250.50**	**771.55**
	RNN-FL(baseline)	**0**	0	0	471.04	806.49
	RNN-FL(TRAIN)	**0**	**1**	**1**	**137.50**	**656.79**
space	MLP-FL(baseline)	2	4	6	517.79	782.55
	MLP-FL(TRAIN)	**3**	**6**	**8**	**461.74**	**632.41**
	CNN-FL(baseline)	0	0	0	2045.82	2182.88
	CNN-FL(TRAIN)	**1**	**4**	**7**	**889.74**	**987.69**
	RNN-FL(baseline)	1	3	4	1218.65	1393.48
	RNN-FL(TRAIN)	**2**	**3**	**6**	**680.24**	**739.53**
total	MLP-FL(baseline)	5	23	44	498.89	891.18
	MLP-FL(TRAIN)	**22**	**67**	**89**	**309.39**	**617.50**
	CNN-FL(baseline)	2	7	12	893.78	1181.39
	CNN-FL(TRAIN)	**8**	**28**	**42**	**473.29**	**805.09**
	RNN-FL(baseline)	3	16	25	793.68	1141.87
	RNN-FL(TRAIN)	**12**	**49**	**76**	**354.27**	**622.31**

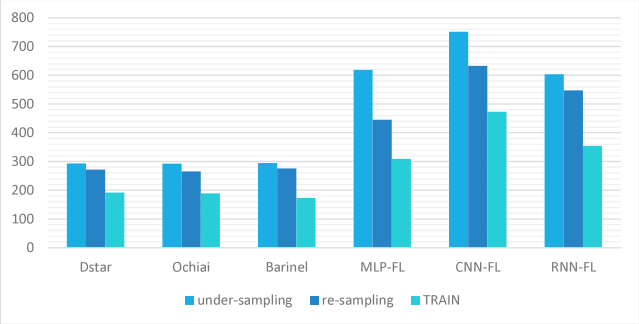

Fig. 11. MFR values of the two data optimization methods and *TRAIN*.

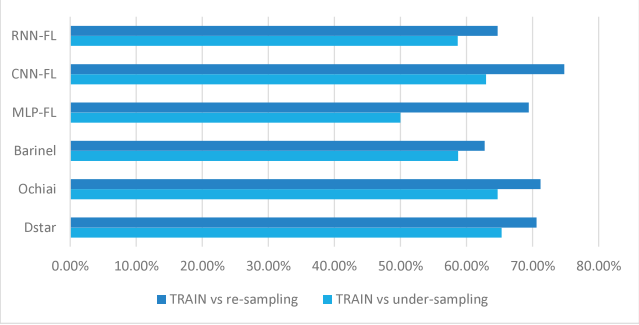

Fig. 12. The RImp of MFR by *TRAIN* over two data optimization methods.

and *TRAIN*. It is evident from the figure that *TRAIN* achieves the most favorable results among these three scenarios.

RImp distribution To comprehensively evaluate the effectiveness of our method *TRAIN*, we utilized RImp as the evaluation metric. Fig. 12 visually represents the distribution of RImp scores for *TRAIN* and two data optimization methods across six FL baselines. As depicted in Fig. 12, all methods achieved RImp scores below 100%, indicating that *TRAIN* improves the localization effectiveness compared to the other methods. The improvement is particularly significant when comparing the under-sampling method to the re-sampling method. Specifically, the percentage of statements requiring examination varied from 49.99% in MLP-FL to 64.62% in Dstar for under-sampling, and from 62.76% in Barinel to 74.80% in CNN-FL for re-sampling. Notably, *TRAIN* achieved minimum saving of 35.38% (100%−64.62% = 35.38%) and maximum saving of 50.01% (100%−49.99% = 50.01%) for under-sampling. In the case of re-sampling, it achieved minimum saving of 25.20% (100%−74.80% = 25.20%) and maximum saving of 37.24% (100%−62.76% = 37.24%).

Statistical comparison To complement the detailed metrics presented earlier, we conducted a WSR Test to provide statistical analysis of *TRAIN* compared to other data optimization methods. Table 8 summarizes the statistical outcomes of *TRAIN* in comparison to data under-sampling and re-sampling methods across three SFL techniques and three DLFL techniques. The "Conclusion" column in Table 8 indicates the conclusion based on the p-value. For example, considering the Ochiai method under the re-sampling scenario, the one-tailed (right), one-tailed (left), and two-tailed p-values are 1.000, 8.70E−06, and 7.56E−05, respectively. According to the definition of the WSR, these values suggest that the MFR value of *TRAIN* is lower than that of the re-sampling method, resulting in an improved outcome. From Table 8, it is evident that *TRAIN* statistically outperforms other data sampling methods in all cases. These results indicate the effectiveness of *TRAIN* in comparison to data under-sampling and re-sampling methods.

Table 6
Statistical comparison for *TRAIN* versus the DLFL baselines.

Comparison		One-tailed(right)	One-tailed(left)	Two-tailed	conclusion
	Chart	1.000	8.47E−08	4.0E−13	**better**
	Time	1.000	7.82E−10	2.79E−06	**better**
	Lang	1.000	1.58E−14	5.45E−12	**better**
	Math	1.000	6.83E−10	7.81E−05	**better**
MLP-FL(TRAIN)	Mokito	1.000	6.96E−14	8.1E−13	**better**
vs	gzip	1.000	2.61E−09	6.91E−13	**better**
MLP-FL	libtif	0.985	6.88E−05	3.36E−05	**better**
	python	1.000	2.73E−11	2.39E−05	**better**
	space	0.998	4.85E−04	5.02E−03	**better**
	total	1.000	5.27E−12	4.37E−16	**better**
	Chart	1.000	3.07E−19	7.73E−15	**better**
	Time	1.000	1.91E−13	3.88E−19	**better**
	Lang	1.000	1.84E−11	5.47E−13	**better**
	Math	1.000	7.39E−10	5.55E−09	**better**
CNN-FL(TRAIN)	Mokito	1.000	3.44E−10	4.26E−10	**better**
vs	gzip	1.000	6.01E−14	8.63E−06	**better**
CNN-FL	libtif	0.997	1.49E−03	7.71E−04	**better**
	python	1.000	8.43E−11	1.89E−07	**better**
	space	1.000	4.01E−16	7.97E−11	**better**
	total	1.000	6.25E−15	4.97E−13	**better**
	Chart	1.000	3.29E−09	8.35E−15	**better**
	Time	1.000	7.95E−09	8.02E−05	**better**
	Lang	1.000	4.8E−09	5.94E−18	**better**
	Math	1.000	6.24E−09	3.55E−05	**better**
RNN-FL(TRAIN)	Mokito	1.000	1.09E−13	2.23E−5	**better**
vs	gzip	1.000	4.94E−09	8.84E−07	**better**
RNN-FL	libtif	1.000	3.91E−5	5.75E−16	**better**
	python	1.000	2.7E−09	7.28E−10	**better**
	space	1.000	3.58E−10	4.37E−11	**better**
	total	1.000	3.99E−13	6.87E−12	**better**

Table 7
Comparisons between *TRAIN* and two data optimization methods for TOP-1, TOP-3, TOP-5, MAR, and MFR.

FL	Scenario	Top-1	Top-5	Top-10	MFR	MAR
Dstar	undersampling	20	86	116	293.55	532.89
	resampling	32	102	128	271.58	397.28
	TRAIN	**38**	**106**	**142**	**191.74**	**369.77**
Ochiai	undersampling	23	94	122	292.45	573.85
	resampling	32	102	129	265.78	410.69
	TRAIN	**38**	**106**	**146**	**189.23**	**370.40**
Barinel	undersampling	21	77	117	294.67	552.67
	resampling	33	96	127	275.75	399.58
	TRAIN	**36**	**100**	**139**	**173.05**	**362.46**
MLP-FL	undersampling	16	46	58	618.89	891.67
	resampling	20	65	77	445.67	836.74
	TRAIN	**22**	**67**	**89**	**309.39**	**617.50**
CNN-FL	undersampling	2	16	28	751.79	1085.75
	resampling	5	23	35	632.78	957.54
	TRAIN	**8**	**28**	**42**	**473.29**	**805.09**
RNN-FL	undersampling	6	26	48	603.81	812.72
	resampling	10	43	62	547.57	754.57
	TRAIN	**12**	**49**	**76**	**354.27**	**622.31**

Summary for RQ3 *In addressing RQ3, we compared TRAIN with two other data sampling methods: data re-sampling and data under-sampling. Based on our comprehensive analysis of all experimental results, we determined that the re-sampling surpasses under-sampling due to the latter's loss of valuable information. Furthermore, it can be observed that TRAIN outperforms both re-sampling and under-sampling methods, demonstrating its superior effectiveness.*

5. Discussion

5.1. Why TRAIN is more effective than the original SFL and DLFL

In this study, we conducted an investigation into the effectiveness of six FL methods when utilizing our proposed method, *TRAIN*. The results obtained from addressing RQ1 and RQ2 exhibited promising outcomes, highlighting the superiority of *TRAIN* over the original SFL and DLFL methods. There are two main reasons why *TRAIN* improves upon the original SFL and DLFL methods. Firstly, CC tests where the execution of faulty program entities coincides with passing test cases adversely affect the effectiveness of FL (Masri et al., 2009). By leveraging an improved cluster analysis method, *TRAIN* effectively identifies CC tests and reverses their corresponding testing results to optimize the trace matrix. This process enhances the fault localization effectiveness. Secondly, in real-world programs, the number of failing tests is often considerably fewer than the number of passing tests. This introduces biases during the learning process and subsequently reduces the accuracy of fault localization models. To address this issue, *TRAIN* employs borderline minority data over-sampling to balance the data, thereby enhancing the effectiveness of fault localization. Prior studies have found that a class-balanced test suite contributes to more accurate fault localization (Yan et al., 2018), and *TRAIN* leverages this knowledge to improve its performance.

5.2. Why TRAIN is more effective than under-sampling and re-sampling FL methods

RQ3 demonstrates that *TRAIN* outperforms both under-sampling and re-sampling techniques in terms of fault localization effectiveness. Under-sampling removes a portion of the majority class samples to balance the input data, which results in the loss of crucial information essential for fault localization (Yan et al., 2018). Conversely, re-sampling randomly duplicates existing failing test cases to achieve data balance. However, the effectiveness of fault localization is limited by two factors. Firstly, re-sampling fails to consider the distribution

Table 8
Statistical comparison of *TRAIN* versus two data optimization methods.

Comparison	One-tailed(right)	One-tailed(left)	Two-tailed	conclusion
TRAIN v.s. Dstar(under-sampling)	1.000	3.62E−05	8.93E−10	**better**
TRAIN v.s. Dstar(re-sampling)	1.000	4.35E−05	8.49E−10	**better**
TRAIN v.s. Ochiai(under-sampling)	1.000	7.46E−05	2.23E−07	**better**
TRAIN v.s. Ochiai(re-sampling)	1.000	8.70E−06	7.56E−05	**better**
TRAIN v.s. Barinel(under-sampling)	1.000	8.92E−05	5.97E−08	**better**
TRAIN v.s. Barinel(re-sampling)	1.000	8.80E−08	1.56E−08	**better**
TRAIN v.s. MLP-FL(under-sampling)	1.000	7.78E−06	6.53E−05	**better**
TRAIN v.s. MLP-FL(re-sampling)	1.000	1.13E−07	2.31E−10	**better**
TRAIN v.s. CNN-FL(under-sampling)	1.000	2.32E−06	4.85E−09	**better**
TRAIN v.s. CNN-FL(re-sampling)	1.000	4.67E−07	3.09E−05	**better**
TRAIN v.s. RNN-FL(under-sampling)	1.000	2.48E−10	3.09E−06	**better**
TRAIN v.s. RNN-FL(re-sampling)	1.000	3.46E−06	7.64E−05	**better**

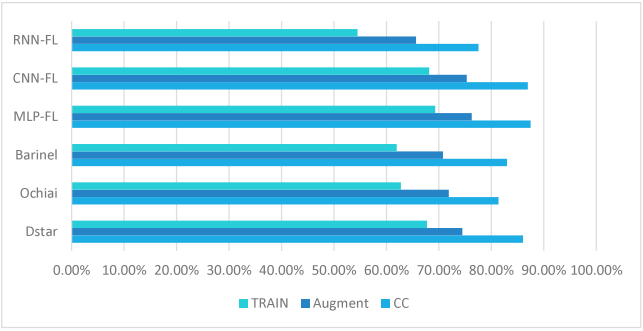

Fig. 13. The comparison of *TRAIN*, *CC* method and *Augment* method over original method under RImp metric.

of minority data during their design, thus disregarding important information from borderline minority data (Han et al., 2005). Secondly, cloning existing data lacks variability in failing tests, potentially leading to over-fitting (Hu et al., 2023). In contrast, *TRAIN* identifies the borderline minority data and synthesizes new data based on them to enhance the effectiveness of fault localization. Moreover, neither of these methods identifies CC tests in the trace matrix, which can degrade the effectiveness of fault localization (Feyzi, 2020). In contrast, *TRAIN* detects the CC tests and reverses the labels corresponding to those tests to optimize the trace matrix. Therefore, *TRAIN* proves more effective than both under-sampling and re-sampling techniques.

5.3. Does each component contributes to the effectiveness of TRAIN?

To evaluate the individual contributions of each step in *TRAIN*, we conducted ablation experiments. These experiments involved implementing the method with only the CC detection stage (*CC* method) and with only the data augmentation stage (*Augment* method). We compared the performance of these methods with that of the original *TRAIN* method using RImp as our metric. By calculating the RImp values of *TRAIN*, *CC* method, and *Augment* method against the original method, we were able to demonstrate the specific contributions of each stage within *TRAIN*.

The results of our ablation experiments are presented in Fig. 13, which provides a comparison of the performance among the three scenarios using the RImp metric. As depicted in the figure, the improvement achieved by utilizing only one stage is relatively small compared to our comprehensive two-stage method. For example, in the case of MLP-FL, the RImp value for the *CC* method is 87.49%, whereas the *Augment* method achieves a RImp value of 76.26%. On the contrary, the RImp value for the complete *TRAIN* method is 69.29%. These findings emphasize that both the CC detection and data synthesis stages play crucial roles in enhancing the effectiveness of *TRAIN*. The *CC* method optimizes the trace matrix by detecting CC tests and reversing their corresponding labels, while the *Synthesis* method utilizes balanced data to improve fault localization.

5.4. Which method fixes best to our method?

From Tables 3 and 5, it can be observed that the RImp values for SFL (Dstar, Ochiai and Barinel) using TRAIN are 93.96%, 81.52% and 73.80%. The average RImp value for SFL is 83.09%. The RImp values for DLFL (MLP-FL, CNN-FL and RNN-FL) using TRAIN are 62.02%, 52.95% and 44.64%. The average RImp value for DLFL is 53.2%. From the average RImp values of SFL and DLFL, it can be found that TRAIN improves the DLFL to a greater extent than SFL. This variance in improvement can be attributed to two primary factors. Firstly, recent research (Heiden et al., 2019) has shown that the Defects4J benchmark tends to over-fit SFL methods such as Dstar, Ochiai, and Barinel. This over-fitting phenomenon makes it challenging for other methods to enhance their effectiveness. Secondly, previous studies (He and Garcia, 2009; Zhang et al., 2021a) have shown that the class imbalance phenomenon of the training set in deep learning causes negative impact on the classifier, which greatly affects the accuracy of the learned localization model. Therefore, DLFL are more sensitive to the imbalanced data, which proves to be more suitable for our method.

Furthermore, when we consider the DLFL techniques, it can be found that our method improves RNN-FL more than MLP-FL for MFR metric. It seems that RNN-FL is best suit for our method. But the MLP-FL achieves the best Top-K metrics (22, 67 and 89) compared to RNN-FL (12, 49 and 76) using our method. A previous study (Parnin and Orso, 2011b) suggested that programmers will only inspect the top few positions in a ranked list. Hence, we conclude that MLP-FL is better suited for our method.

5.5. How k values affect the results?

The classification of the minority data depends on the number of neighbors, denoted as k. To investigate the impact of different k values on the FL results, we employ various k values to classify the minority data. Specifically, we consider k values of 2, 5, 10, 15, and 20, excluding k = 1 since it would result in NOISE and DANGER being considered the same based on their definitions. Fig. 14 illustrates the Rimp scores for each k value in comparison to the original method. As depicted in the figure, all methods achieved RImp scores below 100%, indicating that *TRAIN* enhanced the fault localization effectiveness of the six baselines for all k values. Furthermore, the figure presents two observations. First, the k values have a greater impact on DLFL than SFL, suggesting that DLFL techniques are more sensitive to the generated data compared to SFL. Second, a k value of 5 demonstrates the best performance for FL techniques in these benchmarks, providing guidance for us to select the optimal k value for these programs.

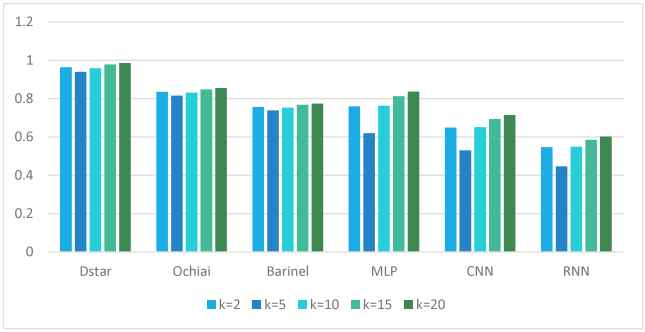

Fig. 14. The comparisons of *TRAIN* over original method under RImp metric for different k values.

6. Threats to validity

Threats to internal validity. The baselines we implemented may not be entirely correct and have inherent randomness. Although we cannot guarantee full compliance of our baseline implementations with those used in their respective experiments, we took steps to mitigate this issue. We utilized GZoltar,[5] a widely used SFL source code, to implement three SFL techniques. Additionally, we employed and enhanced source code from previous studies to implement three DLFL techniques on the source code (Zheng and D. Hu, 2016; Zhang et al., 2019, 2021b). Our team performed double-checks on the implementation and conducted comprehensive tests on the source code. Furthermore, it should be noted that the baselines themselves introduce randomness. When it comes to CC detection, the initial cluster centers for cluster analysis are randomly selected from the trace matrix. This random selection can influence the results obtained using the cluster-based technique.

Threats to external validity. Threats to external validity revolve around the ability to generalize our findings. In our study, we employed MLP-FL, CNN-FL, and RNN-FL techniques, all of which utilize neural networks. It is important to note that neural networks can yield unstable outputs and produce varying localization results during different training intervals. This instability is an inherent limitation of deep learning technology. To enhance the reliability of our findings, we followed a conventional approach. We conducted ten repetitions of the experiments and calculated the average score as the experimental outcome. By doing so, we aimed to mitigate the impact of the variability in the neural networks' outputs.

Another threat to external validity relates to the subject programs used in our study. Although these programs are commonly employed in the software debugging domain and were sourced from real-life development, it is crucial to acknowledge that the experimental outcomes may not be universally applicable. For example, in our method, unexecuted statements by failing tests are removed to obtain a smaller inspecting scope. However, such strategy is only suitable for single-fault scenarios since the selected statements can only reveal its own root cause. If there are multiple faults present in a program, the remaining faults will be disregarded, thereby making our method vulnerable to multiple-fault scenarios. Specifically, when dealing with multiple faults, we encounter two main challenges. The first challenge relates to the partial association of dynamic information with multiple faults, where a failing test case only executes a subset of all the faulty statements associated with multiple faults. Dynamic fault localization (FL) approaches, including our own method, fail to capture the dynamic information of unexecuted faulty statements. Therefore, dynamic methods struggle to effectively identify those faulty statements not executed by the failing test case. The second challenge arises from the intricate effects of multiple faults, such as fault interference and coupling effect (Debroy and Wong, 2009; Fang et al., 2017). Accurate analysis of these effects remains elusive. Dimensionality reduction employed in our method also suffers from this problem and may overlook some of the faulty statements associated with multiple faults. Consequently, our method is ineffective at identifying those faulty statements missed by dimensionality reduction in multiple-fault scenarios. To mitigate this issue, we may leverage clustering technology (e.g., Jones et al., 2007) to transform the context of multiple faults into that of single faults, thus alleviating their combined effect. It would be worthwhile to incrementally expand our study to encompass additional applications, such as programs with multiple faults, to gain further insights.

Threats to construct validity. Threats to construct validity concern the suitability of our evaluation. In our study, we employed commonly accepted metrics such as Top-K, MAR, MFR, RImp, and Wilcoxon-Signed-Rank to evaluate the effectiveness of *TRAIN*. These metrics have been widely adopted in the field, which helps mitigate potential threats to construct validity.

7. Related works

7.1. Suspiciousness evaluation

Researchers develop suspiciousness evaluation models to evaluate the likelihood of a statement being faulty using trace matrix and test results. These models can be classified into two categories:

Spectrum-based fault localization (SFL) SFL leverages the information present in the execution trace matrix and test results to generate a program spectrum that offers an execution profile of program behavior (Wong et al., 2014; Abreu et al., 2009b,a; Jones and Harrold, 2005). The primary objective of SFL is to develop an effective suspiciousness evaluation formula to ascertain the likelihood of a statement being faulty. This typically involves designing an evaluation formula based on the premise that if a statement appears more frequently in failing test cases than passing ones, it should have a higher suspiciousness value of being faulty. Furthermore, SFL is commonly regarded as lightweight, straightforward, and effective. Numerous SFL techniques have been proposed by researchers (e.g., Dstar Wong et al., 2014, Ochiai Abreu et al., 2009b, Barinel Abreu et al., 2009a, Tarantula Jones and Harrold, 2005, Jaccard Chen et al., 2002 and GP02 Tse, 2014), with Dstar (Wong et al., 2014), Ochiai (Abreu et al., 2009b) and Barinel (Abreu et al., 2009a) being the most effective techniques identified by Pearson et al. (2017).

Deep Learning-based Fault Localization (DLFL) Recent studies on Deep Learning based Fault Localization heavily concentrate on neural networks (Lee et al., 2009; Yann Lecun and Eofrey Hinton, 2015; Li et al., 2019; Turaga et al., 2010). DLFL utilizes an execution trace matrix to train models for fault localization that represent statistical correlations between test results (pass or fail) and the execution of various statements in a program (occurrence or non-occurrence) with the use of artificial neural networks. After the training of the model, virtual tests (e.g., [1,0, ...0], [0,1, ...,0], ..., [0,0, ...,1]) are introduced as inputs to the model. In these virtual tests, only one bit is set to one, indicating the coverage of a specific statement. The model then sorts the statements based on their suspiciousness. Several DLFL methods have been proposed by researchers employing different kinds of neural networks such as BP-FL (Maru et al., 2019; Wong et al., 2007), RBF-FL (Wong et al., 2012), MLP-FL (Zheng and D. Hu, 2016), CNN-FL (Zhang et al., 2019), and RNN-FL (Zhang et al., 2021b). In contrast to devising effective neural networks or correlation coefficients, our work focuses on addressing coincidental correctness and class imbalance issues in FL and can be used in conjunction with these FL approaches.

[5] https://gzoltar.com/

7.2. Coincidental correctness (CC)

Researchers in Masri and Assi (2014) proposed a heuristic-based approach to detect CC. They identified a specific set of program entities that are likely to be correlated with CC test cases. This set included elements executed by both failing and a limited percentage of passing test cases. By analyzing the execution of these selected elements in passing tests, they estimated the presence of CC. Moving on, Miao et al. (2013) introduced a cluster-based method for CC detection. Their approach used a coverage matrix as input for cluster analysis, considering test cases grouped within the cluster of failing test cases as CC test cases. Another clustering approach, proposed by Yang et al. (2015), identifies CC test cases in a more restrictive manner. Although cluster-based CC detection techniques have shown impressive performance, selecting an appropriate number of clusters remains a challenge. This issue can be addressed by incorporating supervised algorithms into CC detection. In this regard, Xue et al. (2014) employed ensemble-based support vector machines to detect CC tests. On the other hand, the study conducted in Dass et al. (2020) explored the application of a random forest classifier in CC detection and enhanced its accuracy through ensemble learning methods such as boosting.

7.3. Data augmentation

In addition to the original formulas and neural networks, researchers have proposed various enhanced fault localization techniques based on the input matrix. To tackle the issue of imbalanced test cases, some have suggested the approach of cloning failing test cases (Gao et al., 2013; Zhang et al., 2021a). For instance, Gao et al. (2013) initially introduced a strategy for cloning failing test cases, aiming to create a balanced test suite. They supported their idea with rigorous analysis from a theoretical perspective. Additionally, Zhang et al. (2021a) conducted a large-scale experiment where failing test cases were cloned in SFL techniques. The experimental results aligned with the previous theoretical analysis. However, random re-sampling approaches like the one used in Gao et al. (2013) suffer from the lack of diversity in failing test cases, which can lead to over-fitting. On the other hand, random under-sampling (Wang et al., 2020) may result in the loss of valuable information from the data. In contrast, the *TRAIN* approach synthesizes new failing test cases by utilizing borderline minority data that is susceptible to misclassification. This method aims to balance the input data and provides a specific advantage for fault localization.

8. Conclusion and future work

In this paper, we propose *TRAIN*: a two-stage trace matrix optimization method for fault localization. *TRAIN* incorporates and implements a two-stage approach to optimize the trace matrix. In the first stage, *TRAIN* employs an improved cluster analysis to identify and exclude the CC tests, thereby optimizing the trace matrix. In the second stage, *TRAIN* utilizes data augmentation techniques to enhance the failing test cases and further balance the trace matrix. After two-stage optimization, the trace matrix has fewer CC tests and balanced, which is beneficial for fault localization. We have implemented our method into the FL pipeline and evaluated its performance using datasets from ManyBugs, SIR, and Defects4J. The experimental results demonstrate the superiority of our approach compared to baselines and other two data sampling techniques.

For future work, we plan to expand the scope of our study by incorporating additional subject programs. And we aim to explore more robust CC detection and data augmentation techniques to further improve upon the existing method.

CRediT authorship contribution statement

Jian Hu: Conceptualization, Methodology, Software, Data curation, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the National Key Research and Development Project of China (No. 2020YFB1711900), and the National Natural Science Foundation of China under Grant 61902421.

References

Abdi, H., 2007. The bonferonni and Šidák corrections for multiple comparisons. In: Encyclopedia of Measurement and Statistics. Vol. 3.
Abreu, R., Zoeteweij, P., Gemund, A.J.C.v., 2009a. Spectrum-based multiple fault localization. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 88–99.
Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C., 2009b. A practical evaluation of spectrum-based fault localization. J. Syst. Softw. 82, 1780–1792.
Chawla, N., Bowyer, K., Hall, L., Kegelmeyer, W., 2002. Smote: Synthetic minority over-sampling technique. J. Artificial Intelligence Res. 16, 321–357.
Chen, M., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: problem determination in large, dynamic internet services. In: Proceedings of International Conference on Dependable Systems and Networks. pp. 595–604.
Ciregan, D., Meier, U., Schmidhuber, J., 2012. Multi-column deep neural networks for image classification. In: 2012 IEEE Conference on Computer Vision and Pattern Recognition. pp. 3642–3649.
Dass, S., Xue, X., Siami Namin, A., 2020. Ensemble random forests classifier for detecting coincidentally correct test cases. In: 2020 IEEE 44th Annual Computers, Software, and Applications Conference. COMPSAC, pp. 1326–1331.
Debroy, V., Wong, W.E., 2009. Insights on fault interference for programs with multiple bugs. In: 2009 20th International Symposium on Software Reliability Engineering. pp. 165–174.
Debroy, V., Wong, W.E., Xu, X., Choi, B., 2010. A grouping-based strategy to improve the effectiveness of fault localization techniques. In: Proceedings of International Conference on Quality Software. pp. 13–22.
Donoho, D.L., 2006. For most large underdetermined systems of linear equations the minimal 1-norm solution is also the sparsest solution. Comm. Pure Appl. Math. 59, 797–829.
Eric Wong, W., Debroy, V., Choi, B., 2010. A family of code coverage-based heuristics for effective fault localization. J. Syst. Softw. 83, 188–208.
Fang, C., Feng, Y., Shi, Q., Liu, Z., Li, S., Xu, B., 2017. Fault interference and coupling effect. In: International Conference on Software Engineering and Knowledge Engineering. pp. 501–506.
Feyzi, F., 2020. CGT-FL: Using Cooperative Game Theory to Effective Fault Localization in Presence of Coincidental Correctness. Vol. 25. pp. 3873–3927.
Gao, Y., Zhang, Z., Zhang, L., Gong, C., Zheng, Z., 2013. A theoretical study: The impact of cloning failed test cases on the effectiveness of fault localization. In: Proceedings of 13th International Conference on Quality Software. pp. 288–291.
Gong, C., Zheng, Z., Li, W., Hao, P., 2012. Effects of class imbalance in test suites: an empirical study of spectrum-based fault localization. In: Proceedings of International Conference on Dependable Systems and Networks. pp. 470–475.
Guo, J., Cheng, J., Cleland-Huang, J., 2017. Semantically enhanced software traceability using deep learning techniques. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, pp. 3–14.
Han, H., Wang, W.Y., Mao, B.H., 2005. Borderline-smote: A new over-sampling method in imbalanced data sets learning. In: Huang, D.S., Zhang, X.P., Huang, G.B. (Eds.), Advances in Intelligent Computing. pp. 878–887.
He, H., Garcia, E.A., 2009. Learning from imbalanced data. IEEE Trans. Knowl. Data Eng. 21, 1263–1284.
Heiden, S., Grunske, L., Kehrer, T., Keller, F., Hoorn, A.van., Filieri, A., Lo, D., 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. Softw. - Pract. Exp. 49, 1197–1224.
Heris, S.R., Keyvanpour, M., 2019. Effectiveness of weighted neural network on accuracy of software fault localization. In: 2019 5th International Conference on Web Research. ICWR, pp. 100–104.
Hierons, R.M., 2006. Avoiding Coincidental Correctness in Boundary Value Analysis. Vol. 15. pp. 227–241.
Hu, J., Xie, H., Lei, Y., Yu, K., 2023. A light-weight data augmentation method for fault localization. Inf. Softw. Technol. 157, 107148.

Jones, J., 2004. Fault localization using visualization of test information. In: Proceedings of 26th International Conference on Software Engineering. pp. 54–56.
Jones, J.A., Bowring, J.F., Harrold, M.J., 2007. Debugging in parallel. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. pp. 16–26.
Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 273–282.
Kingma, D.P., Ba, J., 2017. Adam: A method for stochastic optimization. arXiv:1412.6980.
Kochhar, P.S., Xia, X., Lo, D., Li, S., 2016. Practitioners' expectations on automated fault localization. In: Proceedings of International Symposium on Software Testing and Analysis. pp. 165–176.
Krizhevsky, A., Sutskever, I., Hinton, G.E., 2017. Imagenet classification with deep convolutional neural networks. Commun. ACM 60, 84–90.
Küçük, Y., Henderson, T.A.D., Podgurski, A., 2021. Improving fault localization by integrating value and predicate based causal inference techniques. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, pp. 649–660.
Lee, H., Grosse, R., Ranganath, R., Ng, A.Y., 2009. Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations. pp. 609–616.
Li, X., Li, W., Zhang, Y., Zhang, L., 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 169–180.
Li, Y., Liu, C., 2012. Using cluster analysis to identify coincidental correctness in fault localization. In: 2012 Fourth International Conference on Computational and Information Sciences. pp. 357–360.
Li, X., Orso, A., 2020. More accurate dynamic slicing for better supporting software debugging. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification. ICST, pp. 28–38.
Li, Y., Wang, S., Nguyen, T.N., 2021. Fault localization with code coverage representation learning. In: Proceedings of the 43rd International Conference on Software Engineering. pp. 661–673.
Li, X., Zhang, L., 2017. Transforming programs and tests in tandem for fault localization. Proc. ACM Program. Lang. 1, 1–30.
Maru, A., Dutta, A., Kulamala, V.K., Mohapatra, D.P., 2019. Software fault localization using bp neural network based on function and branch coverage. Evol. Intell. 14, 87–104.
Masri, W., Abou-Assi, R., El-Ghali, M., Al-Fatairi, N., 2009. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. pp. 1–5.
Masri, W., Assi, R.A., 2010. Cleansing test suites from coincidental correctness to enhance fault-localization. In: 2010 Third International Conference on Software Testing, Verification and Validation. pp. 165–174.
Masri, W., Assi, R.A., 2014. Prevalence of coincidental correctness and mitigation of its impact on fault localization. ACM Trans. Softw. Eng. Methodol. 23.
Miao, Y., Chen, Z., Li, S., Zhao, Z., Zhou, Y., 2013. A clustering-based strategy to identify coincidental correctness in fault localization. Int. J. Softw. Eng. Knowl. Eng. 23, 721–741.
Parnin, C., Orso, A., 2011a. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 199–209.
Parnin, C., Orso, A., 2011b. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 199–209.
Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, pp. 609–620.
Radford, A., Metz, L., Chintala, S., 2016. Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv:1511.06434.
Ruder, S., 2017. An overview of gradient descent optimization algorithms. arXiv:1609.04747.
Sohn, J., Yoo, S., 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. pp. 273–283.
Tang, C.M., Chan, W.K., Yu, Y.T., Zhang, Z., 2017. Accuracy graphs of spectrum-based fault localization formulas. IEEE Trans. Reliab. 66, 403–424.
Tse, T.H., 2014. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. Comput. Rev. 55, 92.
Turaga, S.C., Murray, J.F., Jain, V., Roth, F., Helmstaedter, M., Briggman, K., Denk, W., Seung, H.S., 2010. Convolutional networks can learn to generate affinity graphs for image segmentation. Neural Comput. 22, 511–538.
Wang, H., Du, B., He, J., Liu, Y., Chen, X., 2020. IETCR: An information entropy based test case reduction strategy for mutation-based fault localization. IEEE Access 8, 124297–124310.
Wilcoxon, F., 1944. Individual comparisons by ranking methods. Biom. Bull. Biometr. 1, 80–83.
Wong, W.E., Debroy, V., Gao, R., Li, Y., 2014. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63, 290–308.
Wong, W.E., Debroy, V., Golden, R., Xu, X., Thuraisingham, B., 2012. Effective software fault localization using an rbf neural network. IEEE Trans. Reliab. 61, 149–169.
Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42, 707–740.
Wong, W.E., Zhao, L., Qi, Y., Cai, K.Y., Dong, J., 2007. Effective fault localization using bp neural networks. In: SEKE. Knowledge Systems Institute Graduate School, pp. 374–379.
Xie, X., Chen, T.Y., Xu, B., 2010. Isolating suspiciousness from spectrum-based fault localization techniques. In: Proceedings of the 2010 10th International Conference on Quality Software. pp. 385–392.
Xie, X., Kuo, F.C., Chen, T.Y., Yoo, S., Harman, M., 2013. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In: Proceedings of the 5th International Symposium on Search Based Software Engineering. Vol. 8084. pp. 224–238.
Xie, H., Lei, Y., Yan, M., Yu, Y., Xia, X., Mao, X., 2022. A universal data augmentation approach for fault localization. In: Proceedings of the 44th International Conference on Software Engineering. pp. 48–60.
Xu, X.F., Debroy, V., Wong, W.E., Guo, D., 2011. Ties within fault localization rankings: Exposing and addressing the problem. Int. J. Softw. Eng. Knowl. Eng. 21, 803–827.
Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L., 2005. A brief survey of program slicing. SIGSOFT Softw. Eng. Not. 30, 1–36.
Xue, X., Pang, Y., Namin, A.S., 2014. Trimming test suites with coincidentally correct test cases for enhancing fault localizations. In: 2014 IEEE 38th Annual Computer Software and Applications Conference. pp. 239–244.
Yan, L., Sun, C., Mao, X., Su, Z., 2018. How test suites impact fault localisation starting from the size. IET Softw. 12, 190–205.
Yang, X., Liu, M., Cao, M., Zhao, L., Wang, L., 2015. Regression identification of coincidental correctness via weighted clustering. In: 2015 IEEE 39th Annual Computer Software and Applications Conference. pp. 115–120.
Yann Lecun, Y.B., Eofrey Hinton, G., 2015. Deep learning. pp. 436–444.
Zhang, X., He, H., Gupta, N., Gupta, R., 2005. Experimental evaluation of using dynamic slices for fault location. In: Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging. pp. 33–42.
Zhang, Z., Lei, Y., Mao, X., Li, P., 2019. CNN-FL: an effective approach for localizing faults using convolutional neural networks. In: Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering. pp. 445–455.
Zhang, Z., Lei, Y., Mao, X., Yan, M., Xu, L., Wen, J., 2021a. Improving deep-learning-based fault localization with resampling. J. Softw. Evol. Process 33, e2312.
Zhang, Z., Lei, Y., Mao, X., Yan, M., Xu, L., Zhang, X., 2021b. A study of effectiveness of deep learning in locating real faults. Inf. Softw. Technol. 131, 106486.
Zhang, M., Vassiliadis, S., Delgado-Frias, J., 1996. Sigmoid generators for neural computing using piecewise approximations. IEEE Trans. Comput. 45, 1045–1049. http://dx.doi.org/10.1109/12.537127.
Zhang, L., Yan, L., Zhang, Z., Zhang, J., Chan, W., Zheng, Z., 2017. A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization. J. Syst. Softw. 129, 35–57.
Zheng, W., D. Hu, J.W., 2016. Fault localization analysis based on deep neural network. In: Proceedings of Mathematical Problems in Engineering. pp. 1–11.

Jian Hu received the BS, MS, and Ph.D. degrees from National University of Defense Technology in 2009, 2012, and 2016, respectively, all in computer science. He is an associate professor in the School of Big Data and Software Engineering at Chongqing University. He is also a member of the CCF Software Engineering Committee, the CCF Integrated Circuit Design Committee and the CCF Computer-Aided Design and Graphics Committee. His research interests include software engineering, electronic design automation and computer-aided design. He has published more than 20 academic papers in DAC, IST, CODES+ISSS, FCS, ATS, ISQED, JCSC, SoCC, SANER, Journal of Computer-Aided Design and Graphics, and Journal of Computer Research and Development.