A Universal Data Augmentation Approach for Fault Localization

Huan Xie
School of Big Data & Software Engineering, Chongqing University
Chongqing, China
huanxie@cqu.edu.cn

Yan Lei*
School of Big Data & Software Engineering, Chongqing University
Chongqing, China
yanlei@cqu.edu.cn

Meng Yan
School of Big Data & Software Engineering, Chongqing University
Chongqing, China
mengy@cqu.edu.cn

Yue Yu
National University of Defense Technology
Changsha, China
yuyue@nudt.edu.cn

Xin Xia
Software Engineering Application Technology Lab, Huawei
China
xin.xia@acm.org

Xiaoguang Mao
National University of Defense Technology
Changsha, China
xgmao@nudt.edu.cn

ABSTRACT

Data is the fuel to models, and it is still applicable in fault localization (FL). Many existing elaborate FL techniques take the code coverage matrix and failure vector as inputs, expecting the techniques could find the correlation between program entities and failures. However, the input data is high-dimensional and extremely imbalanced since the real-world programs are large in size and the number of failing test cases is much less than that of passing test cases, which are posing severe threats to the effectiveness of FL techniques.

To overcome the limitations, we propose **Aeneas**, a universal data augmentation approach that gener**A**t**e**s sy**n**thesized failing t**e**st cases from reduced fe**a**ture **s**pace for more precise fault localization. Specifically, to improve the effectiveness of data augmentation, **Aeneas** applies a revised principal component analysis (PCA) first to generate reduced feature space for more concise representation of the original coverage matrix, which could also gain efficiency for data synthesis. Then, **Aeneas** handles the imbalanced data issue through generating synthesized failing test cases from the reduced feature space through conditional variational autoencoder (CVAE). To evaluate the effectiveness of **Aeneas**, we conduct large-scale experiments on 458 versions of 10 programs (from ManyBugs, SIR, and Defects4J) by six state-of-the-art FL techniques. The experimental results clearly show that **Aeneas** is statistically more effective than baselines, *e.g.,* our approach can improve the six original methods by 89% on average under the Top-1 accuracy.

CCS CONCEPTS

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

*Yan Lei is the corresponding author.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510136

KEYWORDS

Fault Localization, Imbalanced Data, Data Augmentation

ACM Reference Format:
Huan Xie, Yan Lei, Meng Yan, Yue Yu, Xin Xia, and Xiaoguang Mao. 2022. A Universal Data Augmentation Approach for Fault Localization. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510136

1 INTRODUCTION

1.1 Preliminary

Software debugging is a painstaking but important job for developers, which requires a significant level of time and energy. To reduce the cost, various *Fault localization* (FL) techniques [21, 22, 30, 32, 36, 44, 51, 55, 67] have been proposed during the past several decades. FL techniques provide automated ways to assist developers in locating buggy lines that may cause unexpected outputs.

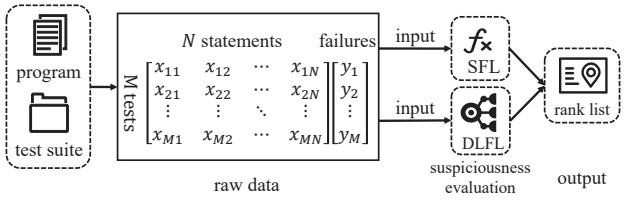

Figure 1: The overall workflow of FL.

Figure. 1 shows the traditional workflow of FL. The **raw data**, which includes the coverage matrix and failure vector, are collected from runtime information when the program is executing the test cases in the test suite. In the code coverage matrix, the rows and columns correspond to the test cases and source code statements, respectively. Each cell (*i.e.,* denoted as x_{ij}, where i means i-th test case, j is the j-th statement) is assigned with the value of 1 if the j-th statement is executed in the i-th test case [1], and with the value of 0, otherwise. For each test case, there is a testing result, which is marked as y_i in Figure 1, where i means i-th test case. The value of 0 will be assigned if the program functions correctly, and with

[1]We can also assign a decimal number to the values of the elements of a vector to represent different weights (*e.g.,* [28, 29]).

the value of 1, otherwise. As we obtained the **raw data**, they serve as the universal input for many FL techniques, such as the most popular ones of *spectrum-based fault localization* (SFL) [2, 36, 55] and *deep learning-based fault localization* (DLFL) [67, 69, 71].

SFL, as a classical localization approach, devises ingenious formulas that could assign each line of code a suspiciousness value. Many formulas of SFL originate in the statistics, coefficient, or probability related to the **raw data** [56]. Different from the intuitive SFL, DLFL utilizes the learning ability of deep learning to construct a localization model for each program, expecting the model could reflect the complex nonlinear relationship between the statements and test results. After the training process ends, a set of virtual test cases in which each test case only covers one single statement will be fed into the model [34, 54, 57, 64, 67, 70, 71]. The output of the model is considered to be the suspiciousness of the corresponding covered statement. Actually, the effectiveness of the model is closely related to the sample data (*i.e.,* the **raw data**). After we obtain the suspiciousness of each statement from either SFL or DLFL, we can thus rank the statements in descending order of their suspiciousness.

1.2 Motivation

From the pipelines of SFL and DLFL, we can observe that the effectiveness of SFL and DLFL heavily relies on the statistical information and quality of the input **raw data**. Nevertheless, the quality of **raw data** poses severe threats due to the class-imbalanced test suite in real-world programs. As presented in Table 1, we list the number of total test cases and that of failing test cases for all faulty versions of a subject program in column '# TT' and column '# FT', respectively. The column 'FT/TT' calculates the ratio of the number of failing test cases over that of all test cases. The values in column 'FT/TT' indicate that the failing test cases only take up a small proportion of all test cases (*i.e.,* less than 3% for most programs). In other words, we face a between-class imbalance problem rooted in the nature of real-world test suite. Furthermore, previous studies have found that a class-balanced test suite is useful for fault localization [13, 65].

Table 1: Statistics of test cases and statements in representative benchmarks.

program	# TT	# FT	FT/TT	# TS	# BS	BS/TS
gzip	30	5	**16.67%**	8586	25	**0.29%**
libtiff	219	18	**8.22%**	29377	1571	**5.35%**
python	1,240	4	**0.32%**	14663	72	**0.49%**
space	461,890	71,935	**15.57%**	123,867	101	**0.08%**
Chart	4,799	92	**1.92%**	104738	64	**0.06%**
Closure	445,683	350	**0.08%**	2199631	431	**0.02%**
Math	18,245	176	**0.96%**	241423	407	**0.17%**
Lang	6,095	124	**2.03%**	52289	251	**0.48%**
Time	68,173	76	**0.11%**	139056	114	**0.08%**
Mokito	25,500	120	**0.47%**	67553	140	**0.21%**

TT: Total Test cases, FT: Failing Test cases, TS: Total executable Statements, BS: Buggy Statements.

However, to the best of our knowledge, existing SFL and DLFL techniques just take raw data as input and rarely take the imbalanced data problem into account, which may consequently cause the ineffectiveness or inefficiency of their approaches. In addition, generating test cases directly from inputs is nearly impossible due to the complexity of the program. Thus, there is an urgent need to tackle the class-imbalanced problem from a different perspective.

Recently, the utilization of deep learning techniques in FL indicates that there are distinguished features (*i.e.,* specific statements) that can make a distinction more effectively between passing test cases and failing ones than traditional SFL. Motivated by the existing DLFL techniques, we can take the coverage matrix and failure vector as samples and labels in machine learning domain, respectively. In other words, each test case corresponds to a sample and each sample is composed of features (*i.e.,* statements). In machine learning, data augmentation is a commonly used solution to the problem of limited data [42]. Oversampling augmentation is one of the data augmentation approaches, which creates synthetic instances by generative networks and adds them into the original data. Inspired by the data augmentation approaches, we intend to apply the generative networks to the **raw data**, expecting the synthetic data could help to improve the effectiveness of SFL and DLFL.

When considering the usage of the generative networks, we found another important problem originated from the program size. In common scenarios, real-world programs are large in size but only several lines of codes or even one statement is responsible for program failure. The low ratios between the number of buggy statements and the number of total executable ones in Table 1 (*i.e.,* the column 'BS/TS') confirm the issue. Mapping this situation to feature space, the dimension of the data is high and there is only a relatively small number of distinguished features. The previous study [10] has shown that high-dimensional data in the input space is usually not good for learning due to the curse of dimensionality.

1.3 Contributions

To deal with the above high-dimensionality and class-imbalanced problems for better data augmentation, we propose **Aeneas**, a universal data augmentation approach that gener**A**t**e**s sy**n**thesized failing t**e**st cases from reduced fe**a**ture **s**pace for improving the universal FL input (*i.e.,* the **raw data**). Specifically, **Aeneas** consists of two stages for data augmentation, namely dimensionality reduction and data synthesis.

Stage 1: Dimensionality reduction. A common way to solve the high-dimensional problem is feature selection, which reduces the dimensionality by selecting a subset of features from the input feature set [14]. The dimensionality reduction aims at obtaining reduced feature space for a better representation of the original coverage matrix, while gaining effectiveness and efficiency. To this end, **Aeneas** adopts a revised *principal component analysis* (PCA) [46] for feature selection as the first stage.

Stage 2: Data synthesis. As for the challenge of imbalanced data, a different way of handling this problem is that we can generate synthesized failing test cases. Concretely, each synthesized failing test is an n-dimensional vector with positive label (*i.e.,* value of 1), where the value of n is the number of statements. Thus, **Aeneas** adopts a widely used generative model *conditional variational autoencoder* (CVAE) [45] to generate the specific class of data (*i.e.,* failing test cases) until the class is balanced.

To the best of our knowledge, our study is the first ever to process the raw data from the feature perspective in fault localization. **We**

implement our approach and intend to make it universal to all FL techniques that take the raw data as input. In summary, this paper makes the following contributions:

1. A new perspective. We provide a novel insight that coverage matrix and failure vector in FL are equivalent to samples and labels respectively in machine learning. Thus, we can apply dimensionality reduction, feature selection, and other machine learning algorithms to the coverage matrix.

2. A universal approach. We propose a universal data augmentation approach, which can be useful for better localization effectiveness. We tackle the imbalanced data problem with an idea that generates synthesized failing test cases by using CVAE. Along with this, we alleviate the high-dimensional issue by adopting a revised PCA technique.

3. Large-scale experiments. We evaluate the effectiveness of **Aeneas** across 458 versions of real-world programs and six representative FL approaches (*e.g.,* Dstar [55] and CNN-FL [67]), and the results reflect the improvement of **Aeneas** over original FL techniques and data sampling approaches.

The remainder of the paper is organized as follows. Section 2 introduces the preliminaries of our approach. Section 3 illustrates our methodology in detail. Section 4 evaluates the effectiveness of our approach, while Section 5 and 6 discuss the limitations and related work, respectively. Finally, Section 7 concludes the paper.

2 BACKGROUND

2.1 Conditional Variational Autoencoder

In Section 1, we find that the test suite is imbalanced in real-world programs. Thus, in this paper, we mainly focus on handling the imbalanced data problem. Many studies deal with the class-imbalanced problem by using different kinds of basic data augmentation methods, such as flip, rotation, scale, crop, and translation [8, 26, 40, 48]. Later, researchers take advantages of generative models (*e.g.,* generative adversarial networks, variational autoencoders, and convolutional neural networks) for data augmentation [5, 35, 37, 39, 43, 50, 58–60, 72, 73]. Xian *et al.* [59] use a popular generative model named conditional *generative adversarial network* (GAN) to generate features for zero-shot learning. Xian *et al.* [60] propose a framework that combines GAN with another powerful generative model named *variational autoencoder* (VAE) to learn and generate the convolutional neural network features for any-shot learning. These techniques, essentially, strengthen the features of the small number of classes. Motivated by these works, we are confident of generating synthesized failing test cases from the coverage matrix.

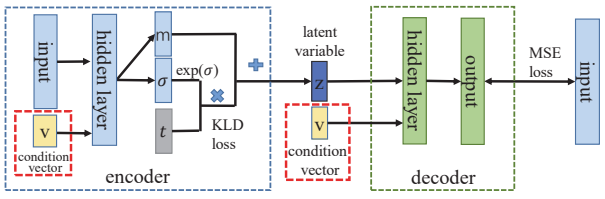

Figure 2: The difference between VAE and CVAE.

Variational autoencoder (VAE) is one of the most popular generative models, which aims at modeling the underlying probability distribution of data so that it could sample new data from the distribution. The structure of VAE network includes two parts, namely encoder and decoder as presented in Figure 2 (the contents of the red dotted rectangle are not included in terms of VAE). The encoder part encodes input data into latent variables and the decoder part generates data from the latent variables. Generally, the decoder part could be used independently of the encoder part as long as the model is well-trained, and we can generate new samples by feeding the decoder a random noise vector. However, the labels of newly generated samples are unknown if we just use a random noise vector as input. In our approach, the data we need is the synthesized failing test cases, but VAE model could not determine the labels of the new samples. *Conditional variational autoencoder* (CVAE) is an extension of VAE, which can control the process of data generation. As shown in Figure 2, CVAE combines the input data and latent variables with labels (*i.e.,* conditional vector v wrapped by red dotted rectangle in Figure 2) in the training process, and then the decoder parts can generate data according to given labels. In other words, the label information is used both in the encoder module for supervision and in the decoder module for guiding the generation.

2.2 Principal Component Analysis

Principal component analysis (PCA) [53] is a technique for exploratory data analysis, dimensionality reduction of large datasets, and predictive models construction [20]. Since PCA is used in exploratory data analysis, we can do data visualization on coverage matrix from a feature's perspective. Concretely, we take the coverage matrix as the input of PCA algorithm and set the number of principal components to two. Finally, we have the data with two dimensions, and we can visualize our data in the coordinate system. Figure 3 shows the visualized plots of some subject programs we used in experiments. There are nine plots in Figure 3 and each plot shows two dimensions (*i.e.,* two principal components) of the coverage matrix after applying PCA. Note that the two dimensions do not correspond to any original statements since new dimensions are obtained by using the linear combination of the original dimensions. In figure 3, the passing test cases are denoted as blue dots, while the orange dots represent the failing test cases. We can observe that the orange dots are well separated from blue dots in these cases, which demonstrates that within the coverage matrix, there are distinguished features that can make a distinction between failing test cases and passing test cases.

From the above exploratory experiment, we notice that PCA is a powerful method that can present intuitive figures of raw data. On the other hand, the reduction process of PCA is to create new variables that do **not** correspond to any original features. Thus, PCA can not be directly used for FL since we need to locate the specific statements (*i.e.,* features), but this information is **not** preserved by PCA. A revised PCA [46], which utilizes the core idea of PCA for feature selection, can fill this gap, and the detail of steps are described in Section 3.2.

3 METHODOLOGY

3.1 Overview

Given a faulty program P with N statements, it is executed by a test suite T with M test cases, which contains at least one failing

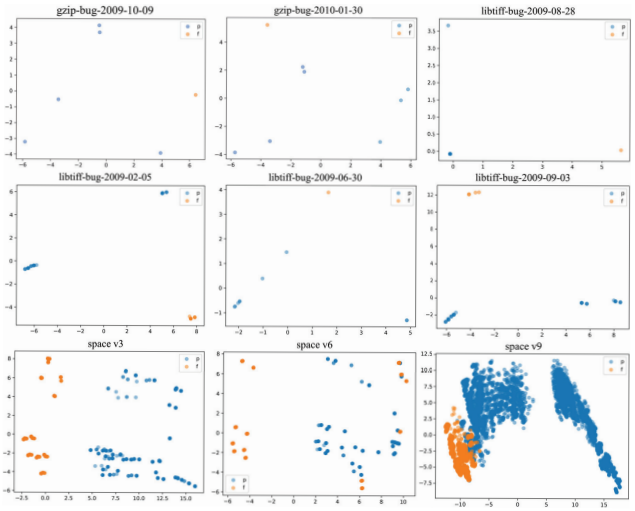

Figure 3: Data visualization for coverage matrix after applying PCA.

test case. According to the runtime information by executing all the test cases, we can obtain the code coverage matrix X with the size of $M * N$, and coverage vector $X(i)$ represents the coverage information of i-th test cases, $i \in \{1, 2, ..., M\}$. By observing the behavior of the program for each test case (*e.g.,* comparing the expected output and the actual one), we could deduce the value of failure vector y, where $y(i)$ means the label of i-th test cases. The code coverage matrix X and the failure vector y are the inputs of our approach.

Our approach includes two stages, namely dimensionality reduction and data synthesis in Figure 4. The first stage is dimensionality reduction, which takes the coverage matrix X as input and outputs reduced coverage matrix X_{reduce} whose size is $M*K$ ($K < N$) after filtering out the potential irrelevant statements. The following data synthesis stage will construct a generative model for the program. In order to model the underlying probability distribution of data, X_{reduce} and condition vector v (*i.e.,* one-hot vector of the corresponding label) are used for the training process. The CVAE model can capture the key latent variables and the decoder can sample specific data by a random noise r and given condition vector v. The generation process ends until the class is balanced, *i.e.,* the number of failing test cases is equal to that of passing test cases. Next, we will depict the above steps in detail.

3.2 Dimensionality Reduction using Revised PCA

We formally present the steps of dimensionality reduction in Figure 4 in this subsection. Algorithm 1 describes the process of feature selection by revised PCA in detail. Besides the coverage matrix X, another two compulsory parameters (*i.e.,* the number of largest eigenvalues m and the number of principal components K) are needed. The algorithm starts with calculating the covariance matrix of X and solves all the eigenvalues and eigenvectors of the covariance matrix (lines 1 - 3). Then, we select the eigenvectors corresponding to the first m largest eigenvalues (line 4), *i.e.,* we obtain a $N * m$ size matrix V consists of m eigenvectors, each eigenvector has N elements. Next, we compute the contribution values (denoted as c) of each feature component by summing m elements in every row of V (lines 5 - 7). According to the contribution value list c, we sort c in descending order and store indexes of them into variable $iContriMax$ (line 8). Now the compulsory data are ready, we can easily select K statements according to $iContriMax$ and add the selected column of X into X_{reduce} (lines 10 - 13). Finally, the algorithm returns X_{reduce} with the size $M * K$ (line 14).

Algorithm 1 dimensionality reduction using revised PCA

Input:
coverage matrix with the size of $M * N$: X
number of largest eigenvalues: m
number of principal components: K
Output:
reduced coverage matrix with size of $M * K$: X_{reduce}
1: $covX$ = covariance matrix of original samples
2: $eigenVec$ = eigenvectors of $covX$
3: $eigenVal$ = eigenvalues of $covX$
4: V = select the $eigenVec$ corresponding to the first m largest $eigenVal$
5: **for** $i = 1; i <= N; i++$ **do**
6: calculate contribution value: $c_i = \sum_{p=1}^{m} |V_{pi}|$
7: **end for**
8: $iContriMax$ = argmax(c)
9: Initialize X_{reduce} as None
10: **for** $i = 1; i <= K; i++$ **do**
11: $selectedStatement$ = select the $iContriMax[i]$-th column of X
12: add(X_{reduce}, $selectedStatement$)
13: **end for**
14: **return** X_{reduce}

The number of principal components (*i.e.,* K) is automatically determined by comparing the number of executed statements with all the faulty historical versions in our datasets. The key idea is: if a faulty version has a larger number of executed statements, the reduction ratio would be higher since there are usually several faulty statements no matter what the size of executed statements is. In detail, we divide all the faulty versions into ten equally sized groups in descending order of the number of executed statements. For the ten groups, we set the reduction proportion into ten values of percentages (from group-1 with 5% to group-10 with 50%, in 5% increments) to determine K.

3.3 Synthesizing Failing Test Cases using CVAE

The data synthesis stage has two main steps, *i.e.,* a training step and a generating step as shown in Figure 4. In the training step, there are two structures called neural networks (NN) encoder and NN decoder (wrapped by the dotted rectangle in Figure 5) because they utilize the learning ability of neural networks.

For the training step (as shown in the top half of Figure 5) of the data synthesis stage, the CVAE model takes the X_{reduce} from the filter stage as training samples. Given any index i ($i \in \{1, 2, ..., M\}$), the CVAE model first converts the corresponding label $y(i)$ into

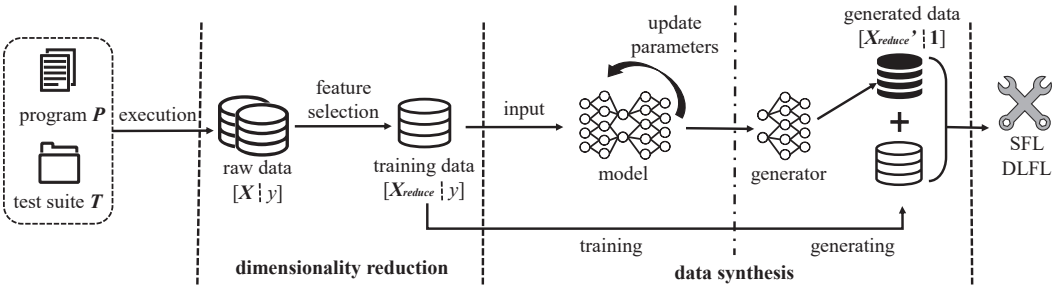

Figure 4: The overview architecture of Aeneas.

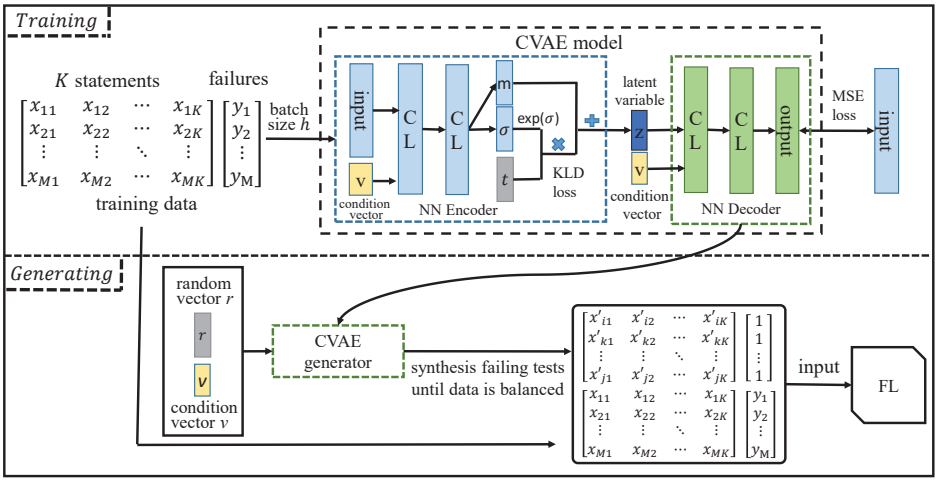

Figure 5: Data synthesis stage of Aeneas.

one-hot vector $v(i)$ and attaches it to the respective coverage vector $X_{reduce}(i)$. Specifically, passing test case (*i.e.,* value of 0) and failing test case (*i.e.,* value of 1) will result in one-hot vectors $[1, 0]$ and $[0, 1]$, respectively. The coverage vector $X_{reduce}(i)$ and the corresponding one-hot vector $v(i)$, as a whole, are the inputs of our model. Followed by the combinational vector $[X_{reduce}(i), v(i)]$, there are two convolutional layers ('CL' in Figure 5). Convolutional layer with the filter can extract features from given data [4], and many studies have fully explored the ability of feature extraction of convolutional neural networks [12, 17, 19, 41, 66]. Since the test cases are independent of each other, we take one-dimensional convolution layers, which the dimension of filter in each convolutional layer is one.

Then, the second convolutional layer of NN encoder generates the mean value m and standard deviation σ, which m controls the value of latent variables and σ is used to compute the weights of the noise vector. The noise vector t comes from randomly sampled from the Gaussian distribution. The encoder finally generates latent variables z by Equation 1 [9].

$$z = m + e^{\sigma} * t \tag{1}$$

To avoid the value of weights e^{σ} infinitely close to 0 (*i.e.,* σ is negative infinity), the CVAE model sets a loss function named KLD loss that could be calculated by Equation 2 [9].

$$KLDloss = -\sum(1 + \sigma - m^2 - e^t)/2 \tag{2}$$

The NN encoder outputs latent variables z and z is the input of the NN decoder. NN decoder also considers the condition vector v that we mentioned above. For each training sample, the NN decoder takes $z(i)$ and $v(i)$ as inputs and outputs the vector $X_{reconstruct}(i)$ that has the same dimension of $X_{reduce}(i)$ through two convolutional layers that are followed by active layers using *ReLU* [18]. Note that both NN encoder and NN decoder could have more than two convolutional layers when considering different program size. By calculating the mean squared error (MSE) of $X_{reduce}(i)$ and $X_{reconstruct}(i)$ by Equation 3 [9], we get another loss of the training process.

$$MSEloss = \frac{\sum_{j=1}^{K}(X_{reduce}(i) - X_{reconstruct}(i))^2}{K} \tag{3}$$

The goal of the training step is to minimize the sum of *KLDloss* and *MSEloss*. We use stochastic gradient descent to update network parameters with batch size h, which means for every step of the training process, a $h * K$ matrix with the number of h corresponding one-hot vectors are fed into the network. In addition, we set the drop rate as 25% to prevent from overfitting [47]. The network is trained iteratively, and the training process will learn a trained model, which can sample new data through the underlying probability distribution.

The generating step (as shown in the bottom half of Figure 5) takes advantage of the NN decoder that is separated from the well-trained CVAE model. Specifically, we can generate synthesized failing test cases X'_{reduce} by feeding the NN decoder a random

noise vector r from Gaussian distribution and condition vector v. The conditional vector v is set to $[0, 1]$ because the class we need is positive. The number of synthesized failing test cases remains to be fixed in this step. Many studies have found that a class balanced test suite is useful for fault localization [13, 65], and algorithms with balanced data should generally surpass these with imbalanced data in performance [15, 25]. Therefore, we synthesize failing test cases until the test suite is class-balanced.

Finally, **Aeneas** takes X_{reduce} and $X^{'}_{reduce}$ as a whole for the next FL techniques (*e.g.,* SFL and DLFL) to address their imbalanced data problem.

3.4 An Illustrative Example

Figure 6 shows an artificial example of a bug. The bug occurs at line 3 in which the number 0 in the 'if' statement should be 6 instead. To locate the buggy line, there exists a category named SFL we mentioned in Section 1. Here we use GP02 [62] for our motivating example. The cells below each statement indicate whether the statement is covered by the execution of a test case or not (1 for executed and 0 for not executed), and the rightmost cells represent whether the execution of a test case is failing or not (0 for passing and 1 for failing). We can observe that the original test suite has six test cases, in which the test t_1 and t_6 are two failing test cases. **Aeneas** will generate two more synthesized failing test cases (*i.e.,* t_7 and t_8) marked with pink in Figure 6.

The rows that start with "GP02", "feature selection", and "GP02 (**Aeneas**)" contain the results of original method, the method using feature selection, and our approach, respectively. With the original test suite, the ranking of suspicious faulty statements of GP02 is $\{s_7, s_8, s_9, s_{12}, s_{10}, s_{11}, s_{14}, s_{15}, s_{16}, s_1, s_2, s_3, s_{13}, s_4, s_5, s_6\}$. The first line marked with blue in Figure 6 shows the results of feature selection using the revised PCA, the sign of × means the corresponding statement is filtered out. With the reduced feature space, the ranking of suspicious faulty statements is $\{s_7, s_9, s_{12}, s_{10}, s_{11}, s_1, s_2, s_3, s_{13}, s_4, s_6, s_5, s_8, s_{14}, s_{15}, s_{16}\}$. With the new test suite generated by **Aeneas**, the ranking of suspicious faulty statements of GP02 is $\{s_8, s_9, s_{12}, s_7, s_1, s_{13}, s_3, s_2, s_{16}, s_{14}, s_{15}, s_{10}, s_{11}, s_5, s_6, s_4\}$. We can observe that GP02 ranks the faulty statements s_3 as the 12th place and after dimensionality reduction, the rank is 8th place. **Aeneas** ranks the faulty statement s_3 as the 4th place, showing better localization results.

4 EXPERIMENT

To evaluate the statistical effectiveness of our approach, we have integrated **Aeneas** into a pipeline that could rank the statements by their suspiciousness. **The code implemented in Python is publicly available at github[2].**

Our experiment was conducted on a 64-bit Linux server with 16 Intel(R) Xeon CPUs and 128G RAM. The operating system is Ubuntu 16.04.3.

4.1 Datasets

We evaluate **Aeneas** on 10 real-world programs with all real faults. As presented in Table 2, we list some statistics of subject programs.

[2]The github repository for this study: https://github.com/ICSE2022FL/ICSE2022FLCode

Table 2: Subject programs

Program	Versions	LoC(k)	Test	Description
gzip	5	491	12	Data compression
libtiff	12	77	78	Image processing
python	8	10	355	General-purpose language
space	38	6	13,585	ADL interpreter
Chart	26	96	2,205	Java chart library
Closure	133	90	7,927	Closure compiler
Math	106	85	3,602	Apache commons-math
Lang	65	22	2,245	Apache commons-lang
Time	27	28	4,130	Standard date and time library
Mokito	38	67	1,075	Mocking framework for Java
Total	**458**	**972**	**35,214**	-

The *space* is collected from SIR [3], the *gzip*, *libtiff* and *python* are collected from ManyBugs [4], while the other programs are from Defects4J [5]. These programs are commonly used for fault localization. Note that it is time-consuming to collect inputs since the programs of Defects4J are large in size, so we reuse the coverage matrix collected by Pearson *et al.* [38].

4.2 Evaluation Metrics

For the evaluation process, we use the following widely used metrics:

- **Number of Top-K [24]:** It is the number of buggy versions with at least one faulty statement that is within the first K position of rank list by a FL technique. In previous study, most respondents view a fault localization as successful only if it can localize bug in the top 5 positions from practical perspective [24]. Following the prior work [32, 44], we assign K with the value of 1, 3 and 5 for our evaluation.
- **Mean Average Rank (MAR) [32]:** For a faulty version, average rank is the mean rank of **all faulty statements** in rank list. MAR is the mean average values for the project that includes several faulty versions.
- **Mean First Rank (MFR) [32]:** It first computes the rank that any of the statements is located **first** for a faulty version. Then compute the mean value of the ranks for the project.
- **Relative Improvement (RImp) [67]:** It is to compare the total number of statements that need to be examined to find all faults using **Aeneas** versus the number that need to be examined by using other fault localization approaches.
- **Wilcoxon signed-rank tests (WSR) [52]:** The metrics above just compare the values by given a specific definition. To evaluate the statistical significance of our results, we adopt Wilcoxon signed-rank tests following prior studies [7, 27, 30, 67].

4.3 Research Questions and Results

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

There are two major popular types of FL: spectrum-based fault localizaiton (SFL) and deep learning-based fault localization (DLFL).

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

Program																			Bug information
S1:Read(a,b,c) S2:d1=0,d2=0,d3=0 S3:if(b < 0){ S4:d1 = b; S5:d2 = c; S6:d3 = a;} S7:else {d1 = b+1; S8: d2 = c+1; S9:if(a < 0){ S10:a = a+c;} S11: else a = a+b; S12: d3 = a+1;} S13:if(c>0){ S14:output(d1);} S15:else {output(d2); S16:output(d3);}													t7 and t8 are new failing tests generated by Aeneas. Then there are 4 passing test cases and 4 failing ones.					S3 is faulty. Correct form: if(b<6){
test	a, b, c	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	result	
t1	-1, 5, 3	1	1	1	0	0	0	1	1	1	1	0	1	1	1	0	0	1	
t2	-2, -7, 5	1	1	1	1	1	1	0	0	0	0	0	0	1	1	0	0	0	
t3	5, -6, -8	1	1	1	1	1	1	0	0	0	0	0	0	1	0	1	1	0	
t4	-5, 8, -8	1	1	1	0	0	0	1	1	1	1	0	1	1	0	1	1	0	
t5	4, 7, 11	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0	
t6	4, 2, -1	1	1	1	0	0	0	1	1	1	0	1	1	1	0	1	1	1	
GP02	susp	6	6	6	4.24	4.24	4.24	8.24	8.24	8.24	6.46	6.46	8.24	6	6.24	6.24	6.24	-	
	rank	10	11	**12**	14	15	16	1	2	3	5	6	4	13	7	8	9	-	
feature selection	result	√	√	√	√	×	√	√	×	√	√	√	√	√	×	×	×	-	
	rank	6	7	**8**	10	12	11	1	13	2	4	5	3	9	14	15	16	-	
t7	-	0.89	0.90	0.90	0.35	-	0.35	0.66	-	0.65	0.38	0.30	0.68	0.87	-	-	-	1	
t8	-	0.96	0.95	0.96	0.34	-	0.38	0.67	-	0.68	0.31	0.38	0.66	0.98	-	-	-	1	
GP02 (**Aeneas**)	susp	9.71	9.71	9.72	5.62	0	5.68	10.90	0	10.90	7.85	7.82	10.93	9.70	0	0	0	-	
	rank	5	6	**4**	11	12	10	3	13	2	8	9	1	7	14	15	16	-	

Figure 6: An example of our approach.

To evaluate the effectiveness of **Aeneas**, we use the six state-of-the-art FL techniques and compare **Aeneas** with them. The six techniques are listed as follows:

- **Dstar**[6] [55],**Ochiai** [1] and **Barinel** [3] are three traditional SFL methods. For more details about them, please refer to [56].
- **MLP-FL** [71], **CNN-FL** [67], and **RNN-FL** [69] are three of the DLFL techniques, which utilize the multi-layers neural network, convolutional neural network, and recurrent neural network for localization. The source code of MLP-FL, CNN-FL and RNN-FL are not public available, so we first acquire the source code of CNN-FL from the authors and then implement MLP-FL and RNN-FL based on the source code of CNN-FL.

We choose MLP-FL, CNN-FL, and RNN-FL as our baseline for the following two reasons. First, our work focuses on statement-level FL whereas many recent DLFL baselines (*e.g.,* DeepFL [30], FLUCCS [44], and TraPT [31]) belong to a different topic, *i.e.,* method-level FL. Second, since our approach only relies on the raw data of test cases, we select baselines that only take the raw data of test cases as inputs without relying on complex source code structure (*e.g.,* AST) analysis, which might increase the difficulty to use it in practice when the size of project is large. Thus, we do not include them (*e.g.,* DEEPRL4FL [32]) as baselines. In such baselines, the data augmentation approach would be another topic which should consider other structures (*e.g.,* data-flow graph and AST) to generate samples.

As a reminder, in our evaluation, when several statements have the same suspiciousness value, we adopt the statement order based strategy [63] that we sort the statements in ascending order according to the line number.

[6]The '*' in Dstar formula is usually assigned to 2.

Table 3: The results of TOP-1, TOP-3, TOP-5, MAR, and MFR by comparison of original method and Aeneas.

metric	scenario	Dstar	Ochiai	Barinel	RNN-FL	MLP-FL	CNN-FL
MFR	origin	448.23	464.89	485.80	1311.63	1816.47	773.57
	Aeneas	**361.08**	**333.82**	**332.54**	**376.03**	**334.78**	**417.56**
MAR	origin	828.84	792.77	807.64	1690.55	2165.98	1167.79
	Aeneas	**714.05**	**631.79**	**611.82**	**638.54**	**594.88**	**767.40**
TOP-1	origin	42	42	38	9	9	31
	Aeneas	**45**	**45**	**42**	**15**	**46**	**45**
TOP-3	origin	98	98	86	33	24	71
	Aeneas	**98**	**99**	**99**	**48**	**98**	**98**
TOP-5	origin	120	122	116	52	30	81
	Aeneas	**127**	**127**	**131**	**73**	**127**	**128**

As presented in Table 3, we list the data under Top-1, Top-3, Top-5, MAR, and MFR metrics. We can observe that **Aeneas** outperforms the original baseline at most cases among six FL approaches. For instance, when examining the Top-1, Top-3, and Top-5 metrics of MLP-FL, the number of bugs that **Aeneas** can locate is 46, 98 and 127 respectively, which means **Aeneas** improves recall at Top-1, Top-3, and Top-5 by 411%, 308%, and 323% in comparison with original MLP-FL. Especially, the improvement observed in the experiments through applying Aeneas on DLFL is more than that on SFL. There are two possible reasons for this. Firstly, the bugs in Defects4J take the most proportion of all bugs and ranks on many faulty versions in Defects4J are already at the top by using Dstar, Ochiai, and Barinel. Secondly, **Aeneas** reformulates the data imbalance problem as data augmentation from the feature perspective in machine learning

domain, which may have more positive influence on deep learning-based approaches.

Note that the experiment results of original DLFL approaches underperform that of the traditional SFL approaches. The work [16] has shown that traditional FL approaches (*e.g.,* Ochiai) perform extremely well on the widely used Defects4J, leading to a benchmark overfitting problem. In fact, for other datasets, DLFL actually performs better [69]. Despite the effectiveness of original DLFL is less than that of original SFL in our experiment, we also take into account various kinds of DLFL approaches considering that the aim of our work is to propose a universal data augmentation approach.

When considering the **MAR** and **MFR** metrics, the rank of **Aeneas** is lower than that of baselines for all six FL techniques, which indicates that we can always locate buggy line first and find all buggy lines with the least effort. Take *Barinel* as an example, the original method will examine 485.80 lines on average to find the first bug in all fault versions (*i.e.,* MFR), while our approach only checks 332.54 lines of code, 68.45% (332.54/485.80) of the original method.

To evaluate the effectiveness of **Aeneas**, we adopt the aforementioned metrics and the results demonstrate our approach outperforms the baselines. However, these metrics fail to get a statistical comparison of our method. Therefore, we adopt the paired Wilcoxon signed-rank tests (WSR), which is a more rigorous and scientific metric at the statistical level. Given independent samples $(x_i, y_i), i \in [1, n]$ from a bivariate distribution (*i.e.,* paired samples), WSR computes the difference $d_i = x_i - y_i$ and gives the p-values. One assumption of the test is that the differences are symmetric. The two-sided test has the null hypothesis that the median of the differences is zero against the alternative that it is different from zero. The one-sided test has the null hypothesis that the median is positive against the alternative that it is negative, or vice versa. If the p-value that calculated by WSR is greater than the given significant level σ, it can be concluded that we can reject the null hypothesis.

Table 4: Statistical comparison of Aeneas and the original methods.

comparsion	greater	less	two-sided	conclusion
Dstar(**Aeneas**) v.s. Dstar	1	1.26E-20	2.53E-20	**BETTER**
Ochiai(**Aeneas**) v.s. Ochiai	1	2.96E-19	5.91E-19	**BETTER**
Barinel(**Aeneas**) v.s. Barinel	1	5.77E-34	1.15E-33	**BETTER**
RNN-FL(**Aeneas**) v.s. RNN-FL	1	8.92E-46	1.78E-45	**BETTER**
MLP-FL(**Aeneas**) v.s. MLP-FL	1	2.90E-55	5.81E-55	**BETTER**
CNN-FL(**Aeneas**) v.s. CNN-FL	1	1.49E-38	2.97E-38	**BETTER**

In our experiment, we use both two-sided and one-sided to check at the σ level of **0.05**. For all faulty versions in the dataset, we use the list of the rank that locates the first bug by using **Aeneas** as the list of x, while y_i is the element in the list of ranks using original baselines. Table 4 shows the statistical results of WSR, in which the columns 'greater', 'less' and 'two-sided' are the p-values. The column 'greater' gives the assumption that the median of $x_i - y_i$ is greater than zero, but the p-values in column 'greater' is greater than σ, which denotes the hypothesis that ranks using **Aeneas** is greater than the ones using original baselines is rejected. That is to say, the number of lines of code to be examined by using **Aeneas** is less than that by using original methods. From the perspective of the statistics, we can observe that **Aeneas** obtains **BETTER** results in six FL techniques.

We also analyze the improvement in terms of MAR/LoC value for a faulty version, where the 'LoC' means executable lines of code for the faulty version. Figure 7 presents the distributions of MAR/LoC values of six FL techniques for all faulty versions. For all six FL techniques, we can observe that **Aeneas** outperforms the original methods, especially DLFL methods.

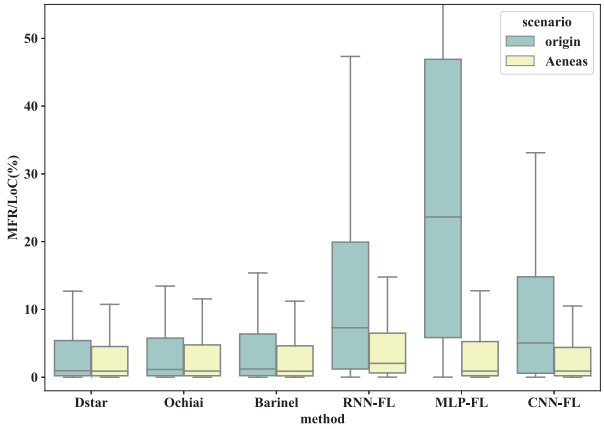

Figure 7: Boxplot of MAR/LoC values of Aeneas and original methods.

Summary for RQ1 *In RQ1, we explore the effectiveness of **Aeneas** over original method. The results show that our method statistically outperforms original method, especially DLFL techniques, which indicates that the neural network model is greatly limited by the imbalanced input data.*

4.3.2 *RQ2.* How effective is Aeneas compared with the data resampling and undersampling approaches?

In addition to the origin methods, we also compare our approach with resampling technique [11, 65, 68] and undersampling technique [49], which aim at obtaining class-balanced data by replicating minority samples and removing the majority samples, respectively. For more details about oversampling technique and undersampling technique, please refer to Gao *et al.* [11] and Wang *et al.* [49], respectively.

As seen in Table 5, **Aeneas** also improves over the resampling and undersampling baselines. From the table, we can clearly find that **Aeneas** can achieve very promising overall fault localization results than undersampling and resampling techniques. For instance, **Aeneas** is able to locate 45, 98, and 127 faults within Top-1, Top-3, and Top-5 metrics when using Dstar. Apart from the Top-K metrics, the MAR and the MFR of **Aeneas** are also the best among the studied approaches.

Although the metrics above can show detailed values, they miss the statistical information. We also use Wilcoxon-Signed-Rank Test to evaluate the effectiveness of **Aeneas** over that of the other data sampling techniques. Table 6 shows the statistical results of **Aeneas** over data undersampling and resampling techniques in three SFL techniques and three DLFL techniques. The 'conclusion' column gives the conclusion according to p-value. For example, in comparison to the resampling technique of Ochiai, the p-value of

Table 5: The results of TOP-1, TOP-3, TOP-5, MAR and MFR by comparison of sampling methods and Aeneas.

metric	scenario	Dstar	Ochiai	Barinel	RNN-FL	MLP-FL	CNN-FL
MFR	re	469.13	482.16	485.74	559.45	492.56	518.63
	under	542.27	532.65	555.53	603.30	976.93	944.11
	Aeneas	**361.08**	**333.82**	**332.54**	**376.03**	**334.78**	**417.56**
MAR	re	867.75	829.70	807.57	909.24	835.36	912.13
	under	968.79	932.62	940.10	988.45	1533.47	1485.77
	Aeneas	**714.05**	**631.79**	**611.82**	**638.54**	**594.88**	**767.40**
TOP-1	re	40	40	38	9	40	40
	under	18	18	16	**18**	14	19
	Aeneas	**45**	**45**	**42**	15	**46**	**45**
TOP-3	re	90	91	86	38	89	90
	under	46	46	38	37	36	46
	Aeneas	**98**	**99**	**99**	**48**	**98**	**98**
TOP-5	re	114	114	115	60	114	114
	under	65	65	58	58	49	64
	Aeneas	**127**	**127**	**131**	**73**	**127**	**128**

're' means resampling and 'under' means undersampling

Table 6: Statistical comparison of Aeneas and the data sampling approaches.

method	comparison	greater	less	two-sided	conclusion
Dstar	undersampling	1	3.05E-40	6.10E-40	**BETTER**
	resampling	1	1.37E-32	2.74E-32	**BETTER**
Ochiai	undersampling	1	2.01E-35	4.01E-35	**BETTER**
	resampling	1	5.64E-34	1.13E-33	**BETTER**
Barinel	undersampling	1	2.85E-40	5.71E-40	**BETTER**
	resampling	1	4.87E-34	9.74E-34	**BETTER**
RNN-FL	undersampling	1	1.43E-50	2.86E-50	**BETTER**
	resampling	1	8.32E-39	1.66E-38	**BETTER**
MLP-FL	undersampling	1	6.81E-44	1.36E-43	**BETTER**
	resampling	1	1.77E-38	3.54E-38	**BETTER**
CNN-FL	undersampling	1	3.23E-23	6.46E-23	**BETTER**
	resampling	1	1.42E-51	2.84E-51	**BETTER**

greater, less, and two-sided are 1, 5.64E-34, and 1.13E-33 respectively. According to the definition of WSR, it means that the MFR value of **Aeneas** is less than that of the resampling technique, leading to a **BETTER** result. From the table, we can observe that **Aeneas** outperforms others data sampling techniques in almost all cases.

Figure 8 visualizes the distributions of MAR/LoC values of six FL techniques for all faulty versions in three scenarios (*i.e.,* the resampling technique, the undersampling technique, and our approach). For all six FL techniques, we can observe that **Aeneas** outperforms other scenarios. Especially in the undersampling technique, the improvement is more significant compared to resampling technique.

Summary for RQ2 *In RQ2, we make comparison between **Aeneas** and other data sampling techniques,* i.e., *data resampling and data undersampling. Based on all experimental results, we can find that resampling technique is much better than undersampling technique since data undersampling technique will drop a lot of useful information. Further, we can observe that **Aeneas** is more effective over both the data resampling and undersampling techniques.*

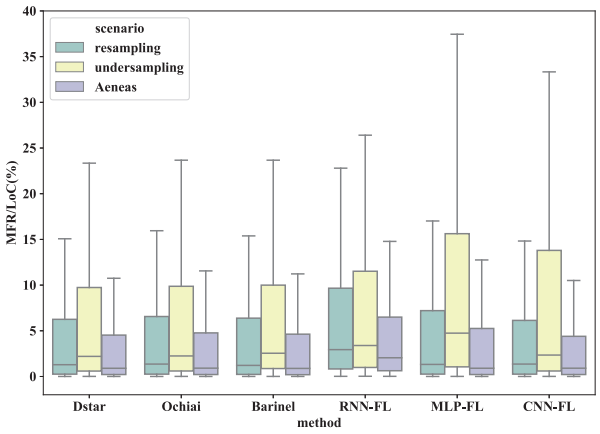

Figure 8: Boxplot of MAR/LoC values of FL techniques in 4 scenarios.

4.3.3 *RQ3.* Is each stage of Aeneas necessary for the capability of Aeneas?

Since **Aeneas** is a two-staged (*i.e.,* dimensionality reduction stage and data synthesis stage) data augmentation approach, it is natural to explore the capability of each stage. Thus, we conduct ablation experiments to explore the contributions of each stage. In order to answer this RQ, we implement the approach with only dimensionality reduction stage and that with only data synthesis stage, and we named them as $\textbf{Aeneas}_{reduction}$ and $\textbf{Aeneas}_{synthesis}$ respectively. We use *RImp* as our metric to demonstrate the contributions of each stage of **Aeneas**. More specifically, we calculate the *RImp* values of **Aeneas**, $\textbf{Aeneas}_{reduction}$ method and $\textbf{Aeneas}_{synthesis}$ method over original method.

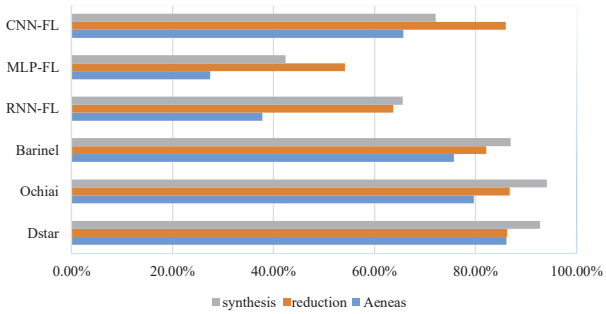

Figure 9: The comparison of Aeneas, $\textbf{Aeneas}_{reduction}$ method and $\textbf{Aeneas}_{synthesis}$ method over original method under RImp metric.

Figure 9 shows the results of the comparison of the three scenarios over original method under *RImp* metric. As we can see in this figure, the improvement of only one stage is relatively small than our two-stage method **Aeneas**. Take Ochiai as example, the *RImp* value of $\textbf{Aeneas}_{reduction}$ is 86.80%, that of $\textbf{Aeneas}_{synthesis}$ is 94.17%, while that of **Aeneas** is 79.69%. The results show that the dimensional reduction stage and data synthesis stage both contribute to **Aeneas**. The reason may be that the reduced feature space from feature selection can make a better representation than raw data. Therefore, the generative model can obtain well-trained parameters that can help to generate more robust negative samples, which are beneficial to locate bugs.

Summary for RQ3 *In RQ3, we explore the capability of each stage of **Aeneas**, our experimental results show that the dimensional reduction stage and data synthesis stage both contribute to our proposed method.*

5 DISCUSSION

5.1 Reasons for Non-improvement

Already/nearly balanced data. We observe that if the test suite of a program is already balanced or nearly balanced, **Aeneas** will have limited effect on the program. *Chart-25* [7] in Defects4J is an illustrative example of such case. This faulty version has four bugs that all belong to *NullCheck* category. Correspondingly, there exist four triggering test cases (*i.e., testDrawWithNullMeanVertical*, *testDrawWithNullDeviationVertical*, *testDrawWithNullMeanHorizontal* and *testDrawWithNullDeviationHorizontal*). The code coverage matrix collected by Pearson *et al.* [38] consists of only nine rows, five for passing test cases and four for failing test cases. Thus, the raw data is nearly balanced in this case. Finally, the original Dstar method just gives the rank of 3133 and both the resampling technique and **Aeneas** can not improve the rank greatly. There are eight (about 2%) versions that are already/nearly balanced in our experiments. For the eight versions, the improvement is less than 3% (under the MFR metric) on average.

Multiple faults. We also take *Chart-25* as the example. For each triggering test case in *Chart-25*, the program failed due to the different root causes. This indicates the features that are responsible for failures are distributed. We conduct an extensive experiment on this faulty version. Concretely, we isolate each failing test case, and then we get five passing test cases and one failing test case as input. The rank of **Aeneas** in such case is 163, revealing a considerable improvement. We further check the original test cases from Defects4J benchmark, and Defects4J reported 3234 test cases for *Chart-25* in total. In other words, the number of lines in the code coverage matrix collected by Pearson *et al.* [38] is inconsistent with that of original test cases. The reason is that they only apply fault-relevant classes to reduce CPU costs. Applying only fault-relevant classes indeed saves CPU costs, nevertheless, it drops a huge amount of useful information that may contribute to the effectiveness of fault localization.

5.2 The Imbalanced Levels Distribution of Subject Programs

In this subsection, we intend to explore the relationship between the extent of improvement and different levels of imbalance.

He *et al.* [15] reported the ratios of 100:1, 1,000:1, and 10,000:1 between majority class and minority class could be viewed as imbalanced data. Since the number of test cases of our subject programs is no more than 10,000, we set three imbalanced levels of subject programs, in the order of their appearance in the sequence: upper level indicates more balanced data.

(1) Ratio that is greater equal than 0.01.
(2) Ratio that is less than 0.01 and greater equal than 0.001.
(3) Ratio that is less than 0.001 and greater equal than 0.0001.

[7] https://github.com/rjust/defects4j/blob/master/framework/projects/Chart/patches/25.src.patch

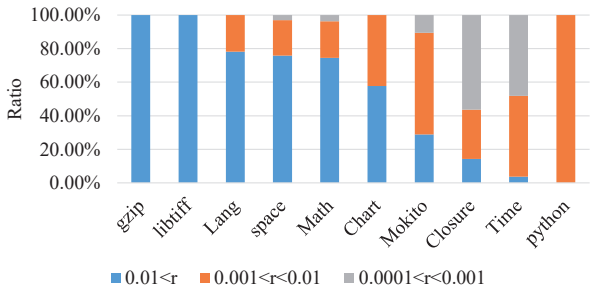

Figure 10: The distribution of imbalanced levels for each subject program.

For each subject program, we count the percentage of each level as shown in Figure 10. In Figure 10, the bar for each subject program indicates the distribution of different imbalanced levels. We count the number of passing test cases and that of failing test cases for a faulty version of a subject program first. Then, we calculate the imbalanced ratios of each faulty version and count the number of ratios in listed intervals. Finally, we could get the percentage of each level for a subject program. Take *Math* as an example, there are 106 bugs and the numbers of ratios of three imbalanced levels are 74.53% (79/106), 21.70% (23/106), and 3.77% (4/106) respectively. The programs in Figure 10 are sorted from left to right in descending order according to imbalanced level (1).

In order to explore the relationship between the different levels of the imbalance and the extent of improvement. We define a metric named Improvement Ratio (ImpR) and the *ImpR* is defined as follows.

$$ImpR = \frac{MAR_{original} - MAR_{\textbf{Aeneas}}}{MAR_{original}} \quad (4)$$

A higher value of *ImpR* shows better improvement of our approach. We calculate the values of *ImpR* based on RNN-FL method and the *ImpR* value increases as the proportion of imbalanced level (1) decreases. In other words, our method is more effective in more imbalanced data.

5.3 Threats to Validity.

Integrity and validity of raw data. We collect the raw data of Defects4J by using the results of Pearson *et al.* [38] for convenience instead of executing test cases to collect coverage matrix and failure vector. But this may introduce the problems of data integrity and validity. As we mentioned before, Pearson *et al.* [38] applied each fault localization technique only to the fault-relevant classes, which directly result in the data integrity problem compared to applying all test cases. We should further spend more CPU costs to collect more complete data before we conduct the experiment. Another problem is data validity. Pearson *et al.* [38] used an improved version of GZoltar [6] that relies heavily on the configurations and the local environment. Even at the same computer with the same configurations, the coverage matrices of two independent runs may differ from each other. A more stable version of GZoltar should be used for collecting raw data.

Implementation of baselines and our method. Our implementation of baselines and **Aeneas** may potentially include bugs. We first implement the Dstar, Ochiai, and Barinel according to the formulas and then test those methods by hand-made test cases for their

correctness. As for RNN-FL, MLP-FL, and CNN-FL, we first acquire the source code of CNN-FL from the authors and then implement MLP-FL and RNN-FL based on the source code of CNN-FL. However, neural networks require many parameters for construction, such as the units of the hidden layer, the number of the layers, batch size, learning rate, and optimizer. That is, the details of RNN-FL, MLP-FL, and CNN-FL may differ from the original paper. Apart from the implementation of baselines, we implement our pipeline of dimensional reduction, test cases synthesis, and fault localization as different modules, which may also contain bugs. To mitigate those threats, our six team members check our code implementation rigorously and make all relevant code publicly available.

Generalizability of the results. Our comparison was only conducted on real faults of Defects4J, SIR, and ManyBugs dataset. Those datasets are widely used in fault localization research. In recent years, Defects4J is a popular dataset for both fault localization and automated program repair. However, in the early years, the fault localization studies frequently used Siemens suite and other C programs with small size [56]. Thus, it is unknown that whether our approach is also effective or not on these programs and other programs with imbalanced test cases. Further, experiments on other datasets should be carried out to migrate this threat.

6 RELATED WORK

Spectrum-based fault localization. In recent years, fault localization (FL) has been intensively studied. Spectrum-based FL (SFL) is one of the typical localization methods. SFL (such as Tarantula [23], Dstar [55], Ochiai [36], Barinel [3], Jaccard [2]) locates bugs from statistical analysis under the assumption that a program entity (*e.g.,* statement, method or class) executed by more failing test cases and not executed by passing test cases is more suspicious. Followed by this, SFL is usually lightweight, straightforward, and effective.

Deep learning-based fault localization. Deep neural networks have shown to be promising in many research areas due to their powerful learning ability. Researchers have reformulated the FL as learning the relationship between program entities and failures. BP neural network-based FL [57], MLP-FL [71] and CNN-FL [67] directly use raw data as training data. DEEPRL4FL [32] devises an enhanced coverage matrix and adopts a test case ordering technique to fully explore the learning ability of the convolutional neural network. Other FL techniques, such as DeepFL [30], FLUCCS [44], and TraPT [31], combine more information for more precise localization. ABLFL [61] takes more comprehensive features, *i.e.,* statistical information (*e.g.,* number of strings, number of integers, and number of operators), semantic information (*e.g.,* code complexity and textual similarity), and dynamic information (*e.g.,* coverage matrix, stack trace, and dynamic program slice), into account for localization. Generally, one method with more information tends to be more effective and less efficient than that with less information. However, the effectiveness also relies on the structure of neural networks and the quality of input data.

Other advanced fault localization techniques. In recent years, there are more advanced FL techniques that leverage various useful approaches. Here are some examples. Automated program repair (APR) has been an essential research topic and FL is a crucial start in APR pipeline. Lou *et al.* [33] improved FL at method level by leveraging the patch execution information of APR techniques as feedback. In this work [27], they combine causal inference techniques and machine learning to estimate the failure-causing effect of statements.

For more information about different types of fault localization techniques, please refer to Wong *et al.* [56].

7 CONCLUSION

In this paper, we propose **Aeneas**, a universal data augmentation approach that aims at improving the effectiveness of fault localization approaches. **Aeneas** is a novel approach to handle the problems of high-dimensional and extremely imbalanced data by feature selection and data synthesis, respectively. Our key ideas include (1) treating coverage matrix and failure vector as samples and labels; (2) reformulating the test cases generation problem as a data augmentation problem; (3) tackling the high-dimensional and class-imbalanced problem by feature selection and data synthesis; More concretely, we use a revised PCA technique for feature selection and a promising generative model CVAE for data synthesis. In the CVAE model, we use convolutional layer as its component due to its effectiveness at extracting features. We have implemented our method and integrated it into the FL pipeline. Further, we evaluated **Aeneas** on the parts of ManyBugs, SIR, and Defects4J and the experimental results show that our method statistically outperforms the baselines and other data sampling techniques. In future work, we intend to use more subject programs and replace CVAE by more powerful generative models.

ACKNOWLEDGMENTS

This work is partially supported by the National Key Research and Development Project of China (No. 2020YFB1711900), the National Defense Basic Scientific Research Project (No. WDZC20 205500308), the Fundamental Research Funds for the Central Universities (No. 2021CDJQY-018), the National Natural Science Foundation of China (No. 62002034), and the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538).

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 88–99.

[4] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. 2017. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*. Ieee, 1–6.

[5] Antreas Antoniou, Amos Storkey, and Harrison Edwards. 2017. Data augmentation generative adversarial networks. *arXiv preprint arXiv:1711.04340* (2017).

[6] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 378–381.

[7] Prantik Chatterjee, Abhijit Chatterjee, José Campos, Rui Abreu, and Subhajit Roy. [n.d.]. Diagnosing Software Faults Using Multiverse Analysis. ([n. d.]).

[8] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 3642–3649.

[9] Carl Doersch. 2016. Tutorial on Variational Autoencoders. (2016), 1–23. arXiv:1606.05908 http://arxiv.org/abs/1606.05908

[10] Richard O Duda, Peter E Hart, and David G Stork. 2001. Pattern Classification: Wiley Interscience. *NY, USA* (2001).

[11] Yichao Gao, Zhenyu Zhang, Long Zhang, Cheng Gong, and Zheng Zheng. 2013. A theoretical study: The impact of cloning failed test cases on the effectiveness of fault localization. In *2013 13th International Conference on Quality Software*. IEEE, 288–291.

[12] Dario Garcia-Gasulla, Ferran Parés, Armand Vilalta, Jonatan Moreno, Eduard Ayguadé, Jesús Labarta, Ulises Cortés, and Toyotaro Suzumura. 2018. On the behavior of convolutional nets for feature extraction. *Journal of Artificial Intelligence Research* 61 (2018), 563–592.

[13] Cheng Gong, Zheng Zheng, Wei Li, and Peng Hao. 2012. Effects of class imbalance in test suites: an empirical study of spectrum-based fault localization. In *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. IEEE, 470–475.

[14] Quanquan Gu, Zhenhui Li, and Jiawei Han. 2011. Generalized fisher score for feature selection. *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence, UAI 2011* (2011), 266–273. arXiv:1202.3725

[15] Haibo He and Edwardo A Garcia. 2009. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering* 21, 9 (2009), 1263–1284.

[16] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, Andre Van Hoorn, Antonio Filieri, and David Lo. 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience* 49, 8 (2019), 1197–1224.

[17] Kai Huang, Ximeng Liu, Shaojing Fu, Deke Guo, and Ming Xu. 2019. A lightweight privacy-preserving CNN feature extraction framework for mobile sensing. *IEEE Transactions on Dependable and Secure Computing* (2019).

[18] Hidenori Ide and Takio Kurita. 2017. Improvement of learning for CNN with ReLU activation by sparse regularization. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2684–2691.

[19] Manjunath Jogin, MS Madhulika, GD Divya, RK Meghana, S Apoorva, et al. 2018. Feature extraction using convolution neural networks (CNN) and deep learning. In *2018 3rd IEEE international conference on recent trends in electronics, information & communication technology (RTEICT)*. IEEE, 2319–2323.

[20] Ian T Jolliffe, Jorge Cadima, and Jorge Cadima. 2016. Principal component analysis : a review and recent developments Subject Areas. *Phil.Trans.R.Soc.A* 374, 20150202 (2016), 1–16.

[21] James A Jones. 2004. Fault localization using visualization of test information. In *Proceedings. 26th International Conference on Software Engineering*. IEEE, 54–56.

[22] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[23] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* IEEE, 467–477.

[24] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[25] Bartosz Krawczyk. 2016. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence* 5, 4 (2016), 221–232.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[27] Yiğit Küçük, Tim AD Henderson, and Andy Podgurski. 2021. Improving fault localization by integrating value and predicate based causal inference techniques. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 649–660.

[28] Hua Jie Lee, Lee Naish, and Kotagiri Ramamohanarao. [n.d.]. Effective Software Bug Localization Using Spectral Frequency Weighting Function. In *Proceedings of the 34th Annual Computer Software and Applications Conference (COMPSAC 2010),* (2010). IEEE, 218–227.

[29] Yan Lei, Xiaoguang Mao, Min Zhang, Jingan Ren, and Yinhua Jiang. [n.d.]. Toward Understanding Information Models of Fault Localization: Elaborate is Not Always Better. In *The 41st Annual Computer Software and Applications Conference(COMPSAC 2017)* (2017). 57–66.

[30] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 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*. 169–180.

[31] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[32] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 661–673.

[33] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.

[34] Abha Maru, Arpita Dutta, K Vinod Kumar, and Durga Prasad Mohapatra. 2019. Software fault localization using BP neural network based on function and branch coverage. *Evolutionary Intelligence* (2019), 1–18.

[35] Seongkyu Mun, Sangwook Park, David K Han, and Hanseok Ko. 2017. Generative adversarial network based acoustic scene training set augmentation and selection using SVM hyper-plane. *Proc. DCASE* (2017), 93–97.

[36] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[37] Hiromitsu Nishizaki. 2017. Data augmentation and feature extraction using variational autoencoder for acoustic modeling. In *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, 1222–1227.

[38] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2016. Evaluating & improving fault localization techniques. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03* (2016), 27.

[39] Luis Perez and Jason Wang. 2017. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621* (2017).

[40] Ikuro Sato, Hiroki Nishimura, and Kensuke Yokoi. 2015. Apac: Augmented pattern classification with neural networks. *arXiv preprint arXiv:1505.03229* (2015).

[41] Giuseppe Scarpa, Massimiliano Gargiulo, Antonio Mazza, and Raffaele Gaetano. 2018. A CNN-based fusion method for feature extraction from sentinel data. *Remote Sensing* 10, 2 (2018), 236.

[42] Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data* 6, 1 (2019), 1–48.

[43] Leon Sixt, Benjamin Wild, and Tim Landgraf. 2018. Rendergan: Generating realistic labeled data. *Frontiers in Robotics and AI* 5 (2018), 66.

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

[45] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. 2015. Learning structured output representation using deep conditional generative models. *Advances in neural information processing systems* 28 (2015), 3483–3491.

[46] Fengxi Song, Zhongwei Guo, and Dayong Mei. 2010. Feature Selection Using Principal Component Analysis. In *2010 International Conference on System Science, Engineering Design and Manufacturing Informatization*, Vol. 1. 27–30. https://doi.org/10.1109/ICSEM.2010.14

[47] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[48] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. 2013. Regularization of neural networks using dropconnect. In *International conference on machine learning*. PMLR, 1058–1066.

[49] Haifeng Wang, Bin Du, Jie He, Yong Liu, and Xiang Chen. 2020. IETCR: An Information Entropy Based Test Case Reduction Strategy for Mutation-Based Fault Localization. *IEEE Access* 8 (2020), 124297–124310.

[50] Qian Wang, Fanlin Meng, and Toby P Breckon. 2020. Data augmentation with norm-VAE for unsupervised domain adaptation. *arXiv preprint arXiv:2012.00848* (2020).

[51] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* (2019).

[52] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[53] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.

[54] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. 2011. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability* 61, 1 (2011), 149–169.

[55] W Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. 2012. Software fault localization using dstar (d*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 21–30.

[56] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[57] W Eric Wong, Lei Zhao, Yu Qi, Kai-Yuan Cai, and Jing Dong. 2007. Effective Fault Localization using BP Neural Networks.. In *SEKE*. Citeseer, 374–379.

[58] Zhanghao Wu, Shuai Wang, Yanmin Qian, and Kai Yu. 2019. Data Augmentation Using Variational Autoencoder for Embedding Based Speaker Verification.. In *INTERSPEECH*. 1163–1167.

[59] Yongqin Xian, Tobias Lorenz, Bernt Schiele, and Zeynep Akata. 2018. Feature generating networks for zero-shot learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 5542–5551.

[60] Yongqin Xian, Saurabh Sharma, Bernt Schiele, and Zeynep Akata. 2019. f-vaegan-d2: A feature generating framework for any-shot learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10275–10284.

[61] Xi Xiao, Yuqing Pan, Bin Zhang, Guangwu Hu, Qing Li, and Runiu Lu. 2021. ALBFL: A Novel Neural Ranking Model for Software Fault Localization via Combining Static and Dynamic Features. *Information and Software Technology* (2021), 106653.

[62] X. Xie, Tsongyueh Chen, Feiching Kuo, and B. Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2013).

[63] Xiaofeng Xu, Vidroha Debroy, W Eric Wong, and Donghui Guo. 2011. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering* 21, 06 (2011), 803–827.

[64] Bei zhang. 2016. Fault Localization Method Based on Enhanced GA-BP Neural Network. In *Proceedings of The fourth International Conference on Information Science and Cloud Computing — PoS(ISCC2015)*. Sissa Medialab, Guangzhou, China, 054. https://doi.org/10.22323/1.264.0054

[65] Long Zhang, Lanfei Yan, Zhenyu Zhang, Jian Zhang, WK Chan, and Zheng Zheng. 2017. A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization. *Journal of Systems and Software* 129 (2017), 35–57.

[66] Mengmeng Zhang, Wei Li, Qian Du, Lianru Gao, and Bing Zhang. 2018. Feature extraction for classification of hyperspectral and LiDAR data using patch-to-patch CNN. *IEEE transactions on cybernetics* 50, 1 (2018), 100–111.

[67] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 445–455.

[68] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Junhao Wen. 2021. Improving deep-learning-based fault localization with resampling. *Journal of Software: Evolution and Process* 33, 3 (2021), e2312.

[69] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Xiaohong Zhang. [n.d.]. A study of effectiveness of deep learning in locating real faults. 131 ([n. d.]), 106486.

[70] Zhuo Zhang, Yan Lei, Qingping Tan, Xiaoguang Mao, Ping Zeng, and Xi Chang. 2017. Deep learning-based fault localization with contextual information. *IEICE Transactions on Information and Systems* 100, 12 (2017), 3027–3031.

[71] Wei Zheng, Desheng Hu, and Jing Wang. 2016. Fault localization analysis based on deep neural network. *Mathematical Problems in Engineering* 2016 (2016).

[72] Fengtao Zhou, Sheng Huang, and Yun Xing. 2020. Deep Semantic Dictionary Learning for Multi-label Image Classification. *arXiv preprint arXiv:2012.12509* (2020).

[73] Xinyue Zhu, Yifan Liu, Zengchang Qin, and Jiahong Li. 2017. Data augmentation in emotion classification using generative adversarial networks. *arXiv preprint arXiv:1711.00648* (2017).