AGFL: A Graph Convolutional Neural Network-Based Method for Fault Localization

Conference Paper · December 2021

DOI: 10.1109/QRS54544.2021.00077

5 authors, including:

Xiaolin Ju
Nantong University
37 PUBLICATIONS **236** CITATIONS
SEE PROFILE

Xiang Chen
Nantong University
125 PUBLICATIONS **2,095** CITATIONS
SEE PROFILE

AGFL: A Graph Convolutional Neural Network-Based Method for Fault Localization

Jie Qian
Nantong University
China
qianjieee@gmail.com

Xiaolin Ju*
Nantong University
China
ju.xl@ntu.edu.cn

Xiang Chen*
Nantong University
China
xchencs@ntu.edu.cn

Hao Shen
Nantong University
China
shenhaohh@gmail.com

Yiheng Shen
Nantong University
China
yiheng.s@outlook.com

2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-5813-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/QRS54544.2021.00077

***Abstract*—Fault localization techniques have been developed for decades. Spectrum Based Fault Localization (SBFL) is a popular strategy in this research topic. However, SBFL is well known for low accuracy, mainly due to simply using a coverage matrix of program executions. In this paper, we propose a method based on graph neural network (AGFL), characterized by the adjacent matrix of the abstract syntax tree and the word vector of each program token. Referring to the Dstar, we calculate the suspiciousness of the statements and rank these statements. The experiment carried on Defects4J, a widely used benchmark, reveals that AGFL can locate 178 of the 262 studied bugs within Top-1, while state-of-the-art techniques at most locate 148 within Top-1. We also investigate the impacts of hyper-parameters (e.g., epoch and learning rate). The results show that AGFL has the best effect when the epoch is 100 and the learning rate is 0.0001. This value of epoch and learning rate increases by 66% compared to the worst on Top-1.**

I. Introduction

Program debugging is one of the essential but exhausting tasks in software development due to the necessary manual activities [1] [2]. In this process, fault localization is the act of identifying the location of faults [3]. Developers often use the experience to determine where these faults are likely to occur, usually by examining statement outputs, inserting breakpoints, and analyzing the results of each test, which follows in an inverse ratio of cost and efficiency.

More and more fault localization approaches that can accurately locate the fault in a specific program were proposed in the past decades [4]. Among these approaches, spectrum-based fault localization(SBFL) is the most widely studied due to its simplifier and intuitive understanding. SBFL counts the coverages of a statement test case that has failed or succeeded. Then, the suspiciousness of each program entity was calculated according to the corresponding risk evaluation formula (e.g., Tarantula [5], Ochiai [6], DStar [7], Jaccard [8], Kulczynski2 [9]). Furthermore, there is also mutation-based fault localization(MBFL), which mutates the source code, obtains the actual influence of the code on the results according to the execution results of test cases, and uses the formula to realize fault localization. (e.g., FIFL [10], Metallaxis [11], and MUSE [12]). Both methods take the execution results of statements into account when calculating the suspiciousness of each program entity, but in some cases, the results are not very satisfying. At the same time, an obvious problem with MBFL is that it is expensive to execute, i.e., sacrificing cost for accuracy.

Recently, many programmers have begun to use deep learning techniques to solve problems in software engineering with the rapid development of deep learning [13]. For example, Liu et al. [14] proposed a back-trace approach of mining non-crashing bugs using behavior graph mining and support vector machine (SVM). Wong et al. [15] propose a RBF neural network-based fault localization technique to assist programmers in locating bugs effectively. Li et al. [16] used the image classification and pattern recognition capability of convolutional neural network (CNN) and applied it to the code coverage matrix [17]. The convolution of CNN uses the kernel to carry out the weighted summation of the central pixel and the set adjacent pixels to form a feature map and realize the extraction of image features. However, CNN applies to image data in Euclidean space, but not in non-Euclidean space structure, which has limitations. Graph convolutional neural networks (GCN) apply to any topology and do not require a fixed number of nodes. GCN can learn both the characteristics of nodes and the association information between nodes.

We proposed a fault localization method based on graph convolutional neural network (AGFL) to address the above issues, enabling us to extract node features in structural diagrams to realize fault localization. According to the source code, to get the structure information, i.e., AST, then traversed the node and through the word2vec [18], get the word vector representation of each node as the node feature. AGFL combines AST structure information and node features to extract features from nodes in the abstract syntax tree and then realizes node classification. We also applied Attention and GCN technologies based on the PyTorch framework. To evaluate our proposed approach, we conducted a fault study on Defects4J. This paper contributes as follows:

- AGFL: A fault localization method based on graph convolutional neural network to predict potential faulty locations via source code information.
- Techniques: Attention mechanism is applied based on graph convolutional neural network to improve the model's accuracy.
- Empirical Study: We empirically evaluate AGFL using 262 real-world faults from Defects4J. AGFL can find 178

2693-9177/21/$31.00 ©2021 IEEE
DOI 10.1109/QRS54544.2021.00077

faults at the Top-1.

The rest of this paper is organized as follows. Section II describes the background of this research topic. Section III presents the proposed model with detailed description. Section IV describes the design of research questions and experimental methods, evaluation metrics. In Section V, we used the Defects4J dataset to evaluate the performance of AGFL, including result analysis and result comparison. Section VI discussed the threats to the validity of our work. Section VII discussed the related work. Finally, We summarized the paper in the last section.

II. Background

A. Abstract Syntax Tree

Abstract syntax tree (AST), which presents the syntax structure of a programming language in the form of a tree, can abstract the syntax information of the source code. Compared to the parse tree, AST is an abstract tree of source code syntax [19]. Because the parse tree contains all the syntax information of the source code, it is a direct translation of the code. The abstract syntax tree ignores some syntax information contained in the parse tree. At the same time, AST also leaves out some unimportant details. For example, parentheses are hidden in a tree's structure and are not represented as nodes. Conditional jump statements, such as the if-then-else statement, are typically represented by nodes with branches. The AST divides the source code into several modules so that the source code structure is clear at a glance, which is conducive to the analysis of the source code. At present, some researchers have applied AST to fault repair [20], code search [21], source code semantic representation [22].

In this paper, we applied the Javalang[1], a third-party open-source library designed by Python language, to analyze the Java source code. Javalang is a pure Python library that provides a Java-oriented lexical analyzer and parser. Using Javalang, the AST node can be divided into several types of nodes. It is using the while loop to achieve 1 to 100, where **While** Statement belongs to the control flow node, Modifier for the declaration node as shown in the figure1. Compared with source code, AST has nodes that do not exist in source code, e.g., Literal and MemberReference, as shown in figure 1. To avoid the influence of the graph structure, we cannot discard this type of node and need to mark its location in the source code with the attributes of its child node. Thus, all nodes in the AST can find their corresponding positions in the source code. When the node is classified as faulty, the location of the fault statement can be directly obtained.

B. Graph Convolutional Neural Network

In 2017, Kipf et al. [23] first proposed a graph convolutional neural network (GCN). GCN combines the characteristics of graph neural networks and convolutional neural networks. Additionally, it extends the convolutional neural network, which is only suitable for Euclidean space, to graphs in non-Euclidean space. GCN is a semi-supervised model that can deal with graph structure. It uses node attributes and node labels to train the model end-to-end. Its essential purpose is to extract the spatial features of topology. As for CNN, which can also extract features from images, it can only process data in Euclidean space, i.e., the data structure should be regular. However, not all pictures are regular; many of them are irregular graph structures or topology structures. In contrast, GCN produces a broader range of applications, which should lead to the advancement of GCN in many applications, e.g., image classification, document classification, and unsupervised learning.

Convolution modes in GCN include spectrum and spatial domain convolution [24]. Spectrum convolution uses the theory of the graph to realize the convolution operation on the topology. The spatial convolution acts on the node's neighborhood, and the node's feature representation is obtained through the aggregation of the node's neighbor.

GCN model is composed of the input layer, hidden layer, and output layer. In the input layer, we take the feature vector and adjacency matrix of the graph as inputs. Then, through multi-layer graph convolution and other operations and activation functions, the representation of each node can be obtained, facilitating node classification. Each hidden layer corresponds to a feature vector, and each row in the matrix is the characteristic representation of a node. After each hidden layer, GCN will aggregate information according to propagation rules to form features of the next layer. The propagation rule of each convolution layer in the hidden layer is defined with equation (1).

$$H^{(l+1)} = \sigma(D^{\frac{-1}{2}}\tilde{A}D^{\frac{-1}{2}}H^{(l)}W^{(l)}) \tag{1}$$

where $\tilde{A} = A + I_N$ is an adjacent matrix with self-connection structures, I_N is an identity matrix; D is a degree matrix of $\tilde{A}$, i.e., $D = \sum_j \tilde{A}_{ij}$; H^l is the matrix of activation in the l layer. In the first hidden layer, $H^0 = X$; $W^{(l)}$ is the weight matrix from the hidden layer to the output layer. $D^{\frac{-1}{2}}\tilde{A}D^{\frac{-1}{2}}$ is the normalized adjacency matrix, $H^{(l)}W^{(l)}$ can be seen that the embedding of all nodes in l layer goes through a linear transformation and then multiplies left with the adjacency matrix. For each node, the feature representation of the node is the result of adding features of the neighboring nodes.

GCN captures spatial correlations by determining topological relationships between nodes and encoding node features. In addition, the input nodes of GCN are variable, and the number of input nodes does not need to be fixed. Therefore, GCN has broad applicability and can be used for nodes and graphs of any topology structure. We use GCN to obtain characteristic information of source code, classify nodes, and then find wrong statements according to faulty nodes.

C. Attention Model

Bahdanau et al. [25] was the first to use an attention mechanism in machine translation. Then the neural network based

[1]https://github.com/c2nes/javalang

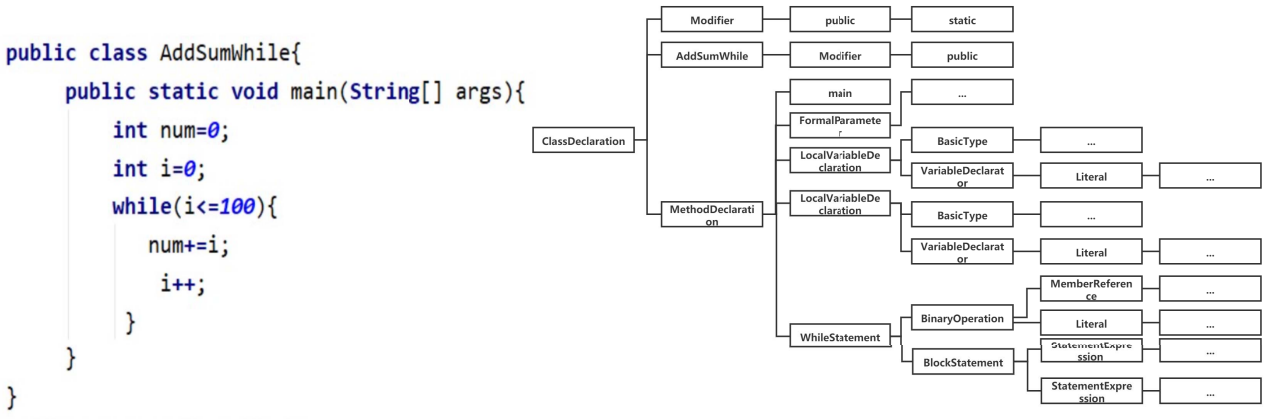

Fig. 1. Motivation code with its abstract syntax tree (AST).

on attention mechanism became a hot spot in neural network research. Existing attention mechanisms can be classified into soft attention, hard attention [25], Local attention [26] and self-attention [27]. We use the Soft Attention model to learn the importance of each node's features so that each node can be classified based on a word vector of overall trends.

$$e_i = W^{(1)}(W^{(0)}H + b^{(0)}) + b^{(1)} \tag{2}$$

$$\alpha_i = \frac{exp(e_i)}{\sum_{j=1}^{n} exp(e_j)} \tag{3}$$

$$C_t = \sum_{i=1}^{n} \alpha_i * h_i \tag{4}$$

The feature of each node is taken as input when the attention mechanism calculates the weight of each hidden state. The output is obtained through two hidden layers, and the Softmax normalized exponential function (2) and (3) calculates the weight of each feature α_i, Where $W^{(0)}$ and $b^{(0)}$ are respectively the first layer of the weights and bias. $W^{(1)}$ and $b^{(1)}$ are the second layer weights and bias. Finally, the feature vector C_t is calculated based on the global variation information, as shown in the equation (4).

GCN needs to compute the entire graph, which leads to the need for large amounts of memory. We use the attention mechanism to reduce the dimension of node characteristics first. Furthermore, the structure of abstract syntax trees is often large and deep, making it is difficult to capture long-distance dependencies. Moreover, the attention mechanism gives importance to the edges between nodes to help the model learn structural information, i.e., each encoded-word vector representation corresponds to a weighted vector that encodes contextual information to capture dependencies at any distance. Therefore, processing node features through an attention mechanism is helpful for GCN to classify nodes.

III. Our Approach

The overall goal of our work is to give an effective solution for fault localization. Since AST can represent the features of a program, we compute the AST of the software under test (SUT). Furthermore, we applied the Attention mechanism and GCN for the evaluation of suspiciousness of each program entity.

The framework of our is shown in Fig. 2. Our approach is mainly composed of three phases. `data processing`, `model constructing`, and `fault localization report`. The first phase extracts the program's features by transforming them into AST and combining it with GraphSmote matrices. The second phase is to construct a prediction model by training with new matrices after GraphSmote. The last phase is to calculate each program entity's suspiciousness and give a fault localization report. Next, we will introduce the implementation of these three phases in sequence.

A. Data processing

In this phase, we check out the source code from Defects4J, the bug line file with the annotated fault information, and convert all the source code to multiple ASTs. There is a set of nodes S in each AST, where each node $s \in S$ corresponds to the token of statements in the source code. Since we use Javalang for tokenizing and AST generation, the tokens will be different from the source code, e.g., **Method Declaration**, **For Statement**, but we can still find its location in the source code according to node information. Then, the adjacency matrix with graph structure information can be obtained by traversing the graph composed of multiple AST.

We take AST as an undirected graph, and the adjacency matrix is symmetric across the diagonal. Suppose there are n nodes in the graph, and the dimension of the adjacency matrix is n, i.e., the adjacency matrix A is A matrix $n * n$, and the values in the matrix are 0 or 1. In the program, erroneous statements are far less than the correct ones. If the data are trained directly, the results will be biased to the

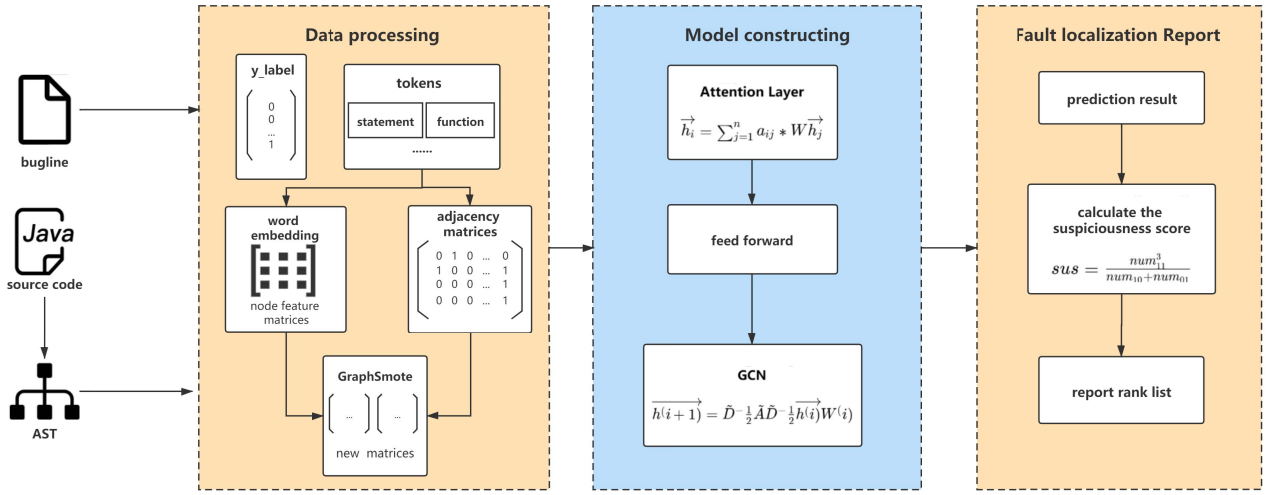

Fig. 2. Framework of our approach

category of correct. To avoid such problems, we need to preprocess the data to solve the problem of category imbalance. Category imbalance refers to the fact that the number of samples of some classes is much less than that of others [28]. In classification problems, it is usually assumed that the same proportion of training is given to different categories. In practical applications, such an idealized situation rarely exists, and it is often possible that there are too many sample data of one or some categories, resulting in an unbalanced distribution of training sample categories. Assuming that there are two categories of samples, the ratio of positive and negative samples reaches 30:1. The model only needs to predict the new sample as a positive category, and its accuracy can reach high. However, such models are of no value for categorical prediction.

Up to now, the class imbalance can be classified into two types: under-sampling and oversampling. Under-sampling refers to sampling a category with many samples to reduce the number of samples in that category. In this way, the number of categories can be kept close, and the training results will be better. Oversampling is sampling a small number of data to increase the number of data. However, in the oversampling process, some important information is discarded due to the random discarding of samples, which is not conducive to the prediction of results. Meanwhile, to be close to the number of samples with fewer categories, the training data are significantly reduced, which leads to the phenomenon of overfitting, e.g., the training data can get better results, but the data can not be well fitted on testing or validation data. R Mohammed et al. [29] compared the performance of the two methods in machine learning. The results show that oversampling performs better than undersampling and gets higher scores in different evaluation metrics. Therefore, we choose oversampling to process the obtained data.

Since SMOTE [30] is the classic oversampling algorithm, using the K-nearest neighbor to randomly select from its nearest neighbor for each data in a few classes, and a random point on the line between these two is chosen as the resultant new data. This process is based on the Euclidean distance between different kinds of samples, which only applies to the data in Euclidean space and cannot be directly used in the graph structure in this paper. Zhao et al. [31] proposed the GraphSMOTE method, like SMOTE idea, using the interpolation method on the GNN-based feature miner, to create a few category nodes. Furthermore, the edge generator predicts the link between synthetic nodes, which is convenient for the subsequent node classification. So we use GraphSMOTE to deal with the data so that the nodes are evenly spread.

The source code contains various symbols and grammar information, so we use all tokens obtained through traversal as a corpus for training and then generate the word vector of each node. Word2vec can be used to represent lexical information for each node. For each node featuresX_i, $i \in n$, X_i is set to length 100. Because the number of tokens with faults is much smaller than the number of correct tokens, e.g., the ratio of two categories in Time reaches 1200:1, we treat the adjacent matrix A and characteristic X by GraphSMOTE. Finally, we get the new adjacent matrix $\overline{A}$ and characteristic matrix $\overline{X}$, where A is $n * n$ matrix and X is $n * 100$ matrix.

B. Model constructing

Due to the large structure of the diagram, we set the number of hidden layers in the attention layer to 2 and the output dimension to 20. After the attention layer, while information is aggregated, the dimension of each node can be reduced to 20, i.e., X is a $N*20$ matrix, which reduces the space pressure for GCN in the next step. Then the processed node feature $\overline{X}$ and the adjacent matrix $\overline{A}$ are feedback to GCN. LeClair et al. [32] have shown that two-level graph convolution is the best setting for obtaining code information, so we set the convolution level to 2. After two layers of convolution 1, node information is further aggregated, and semantic information is extracted. The

activation functions of these two layers are ReLU and Softmax, respectively. SoftMax can obtain two kinds of prediction probabilities, and the overall forward propagation formula is (5).

$$Z = Softmax(\tilde{\overline{A}}ReLU(\tilde{\overline{AX}}W^{(0)})W^{(1)}) \quad (5)$$

In the training process, we randomly divided the data into training, validation, and test sets to prevent the phenomenon of over-fitting. Firstly, the training set is used to fit the model, and then the parameters in GCN are adjusted, and the validation set preliminarily evaluates the model. Based on the evaluation results, the loss function calculates the error and updates all parameters. Model training aims to minimize the errors between the real and predicted values of nodes. For the binary classification problems, we use cross-entropy as the loss function. The cross-entropy loss function is calculated as the formula (6). In addition, after 10 epochs at a time, we test the model's ability to generalize when it is actually used on a test set.

$$\mathcal{L}(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{y}_i^{(1)}\log\hat{\mathbf{y}}_i^{(1)} + \mathbf{y}_i^{(2)}\log\hat{\mathbf{y}}_i^{(2)}\right) + \frac{\lambda}{2}\|\theta\|_2^2 \quad (6)$$

C. Fault localization Report

After the second stage, we obtain the prediction results of the model for nodes, i.e., the model outputs $n * 1$ matrix, and the values in the matrix are presented as 0 or 1. We refer to the method of Dstar [7] when dealing with suspiciousness of the statement. Dstar defines the formula (7) according to the number of successful or failed test cases in the statement, where a_{11}indicates the number of failed test cases that covered the statement; a_{01} indicates the number of failed test cases that did not cover the statement; a_{10} indicates the number of successful test cases that covered the statement.

$$susp(s) = \frac{a_{11}^{*}}{a_{01} + a_{10}} \quad (7)$$

We respectively define a_{11}, a_{01}, a_{10} as the number of faulty nodes that predicated successfully, the number of correct nodes that did not predicate successfully, the number of faulty nodes that did not predicate successfully.

IV. Empirical study

A. Research Questions

To evaluate the effectiveness and efficiency of our approach (AGFL), we implemented it in a prototype tool and applied the tool to 5 subject programs and corresponding faults. In our empirical study, we want to investigate the following research questions:

RQ1: How does AGFL perform in locating real faults compared with state-of-the-art techniques?

This question helps us to understand the performance of widely-used techniques. We adopt Mean Average Rank (MAR) and Top-k metrics, which are widely used in previous studies for comparing different fault localization approaches.

RQ2: How much does SMOTE contribute to fault localization?

Class imbalance issues can significantly influence the classifier's performance in those minority classes. This question considers the performance impact of pre-processing the data.

RQ3: How do different epochs and learning rates impact AGFL results?

In our empirical study, we set the epoch for all projects to 100. However, the optimal epoch may differ for data in different projects. This question considers a specific way of combining different epochs and learning rates and evaluates the technique's performance.

B. Experimental setup

In order to answer these RQs, we conducted an empirical study on five subjects of real-world faults from Defects4J (v2.0.0) [2]. All five subjects total have 262 real faults. TABLE I illustrates the details of Chart, Lang, Math, Mockito, and Time used in the experiment. The second column in the table is the program name, the first column is its short name, the third column is a count of the number of lines of code, the fourth column is the number of faults versions provided, and the last column is the number of faults IDS that have been deprecated. Defects4J provides the program version of the bug and fix for each fault and indicates which lines in the program have faults. In projects with multiple faults, we select the location of the first one without loss of generality.

TABLE I
CHARACTERISTICS OF PROJECTS

Identifier	Project name	LOC(k)	Number of bugs	deprecated bugs
Chart	jfreechart	96	26	None
Lang	commons-lang	22	65	2
Math	commons-math	85	106	None
Time	joda-time	28	27	1
Mockito	mockito	23	38	None
Total		254	262	3

When generating the word vector, we use Word2Vec with the Skip-Gram algorithm to obtain the embedding word vector and set the embedding size to 100. Referring to GraphSMOTE [31], we set the parameters in GCN as learning rate 0.0001, the number of hidden layer nodes = 64, dropout = 0.1, epoch = 100. Because we randomly divided the data set, we ran each project ten times and took the average of ten times as the final result. In Chart and Math, limited memory makes it impossible to process the entire graph structure. We divided each project into multiple parts. The parameters of the model were saved at the end of each training. When the next part was run, the parameters of the previous part were loaded to finish the training of the model.

We run our implementation on a machine with Intel Xeon Platinum 8260M CPU @ 2.30GHz and 86GB RAM for faster computation.

[2]https://github.com/rjust/defects4j

C. Evaluation metrics

Many metrics are applied to evaluate the accuracy and effectiveness of software fault localization techniques. In this paper, the performance of AGFL is evaluated by MAR and Top-K.

1) Mean Average Rank: Mean Average Rank (MAR) is the average ranking of all faults. For each subject, MAR is the average of all its rankings.

2) Recall at Top-k: Top-k indicates the number of faults found in the first K statements of the sorting result. The higher the top-k value is, the more faults can be found by the method when examining the same number of statements. Kochhar et al. [33] conducted an empirical study on fault localization, and the result shows that 73.5% of researchers consider top-5 to be the minimum standard for the success of the definition method. Therefore, n can be set as 1,3,5 in this paper, respectively.

V. Result analysis

A. Answer for RQ1

To evaluate the performance of AGFL for fault localization accurately, we compare our approach with the following five methods:

1) DeepFL [34]: using multi-layer perceptrons and treated suspiciousness value, fault proneness, and textual similarity as the characteristics of the model to achieve the location of faults
2) FLUCCS [35]: extending SBFL with code change metrics and applying genetic programming, support vector machines to sort suspiciousness
3) TraPT [36]: using learning-to-rank technique and combining with SBFL and MBFL techniques.
4) Ochiai [6] and Dstar [7]: two classic spectrum-based fault localization methods which are widely used in previous work [3].

For RQ1, we use MAR and Top-K to evaluate the accuracy of related methods. TABLE II illustrates detailed experimental results.

TABLE II
Effectiveness of AGFL and compared techniques

Subjects	Techniques	Top-1	Top-3	Top5	MAR
Chart	Ochiai	6	14	15	4.11
	FLUCCS	15	19	20	4.3
	TraPT	10	15	16	5.7
	Dstar	5	16	19	9.51
	DeepFL	12	20	20	9.23
	AGFL	14	15	15	7.26
Lang	Ochiai	24	44	50	2.53
	FLUCCS	40	53	55	3.63
	TraPT	42	55	58	3.18
	Dstar	24	49	59	5.01
	DeepFL	46	54	59	4.6
	AGFL	42	46	46	3.35
Math	Ochiai	23	52	62	4.84
	FLUCCS	48	77	83	5.66
	TraPT	34	63	77	6.84
	Dstar	24	63	75	11.72
	DeepFL	63	85	91	11.35
	AGFL	75	84	87	7.88
Time	Ochiai	6	11	13	12.62
	FLUCCS	8	15	18	11.9
	TraPT	7	13	16	13.19
	Dstar	6	11	12	18.87
	DeepFL	13	17	17	18.26
	AGFL	15	15	15	2.26
Mockito	Ochiai	7	14	18	13.78
	FLUCCS	7	19	22	18.63
	TraPT	12	20	22	26.97
	Dstar	7	16	19	24.77
	DeepFL	12	19	22	22.73
	AGFL	32	32	32	1.08
Overall	Ochiai	66	135	158	37.88
	FLUCCS	118	183	198	44.12
	TraPT	105	166	189	55.88
	Dstar	66	155	184	69.88
	DeepFL	146	195	209	66.17
	AGFL	178	192	195	21.83

As shown in the TABLE II, column 1 lists the project names in Defects4J; Column 2 lists all the methods of comparison; The remaining columns show the Top-k (k = 1, 3, 5) and MAR evaluation results. From the TABLE II, we can see that AGFL could locate 178 of the 262 faults at top-1, while DeepFL could locate only 146. In addition, the MAR of AGFL is 57.6%, 49.4%, 39%, 31.2%, 32.9% of other methods, respectively. That is to say, our method, AFGL, can assign a high ranking to faulty statements. Overall, AGFL performed better than other methods, especially on Mockito and Time. In other cases, AGFL's performance is close to that of other methods but not as good as DeepFL. The root causes of this problem can be summarized as follows:

First, the subject Chart is a library diagram-drawing class. By comparing the source code with the repaired code of Chart, we found that most of the faults are fixed by adding statements rather than modifying tokens based on the original statements in Chart. Therefore, our model, which is trained by tokens, cannot identify whether tokens have faults correctly. Second, we only run the subjects in parts, e.g., in Math, we trained in groups of 20 faults by saving the model parameters from the previous group. Although the model parameters are constantly optimized, there may have the problem of local optima, which will eventually affect the model effectiveness. The above two reasons can better explain why AGFL loses its effectiveness compared with the other five fault localization approaches.

Summary for RQ1: AGFL can localize more faults than other compared methods. Significantly, the accuracy of AGFL is better than other methods on Top-1, and the MAR of our method is also reduced.

B. Answer for RQ2

For RQ2, this paper uses Top-k to analyze the effect of SMOTE method. Fig. 3 shows the number of nodes before and after data processing. For each subject, the number of nodes increases by 10% on average. TABLE III shows the results before and after data processing, where n represents nodes

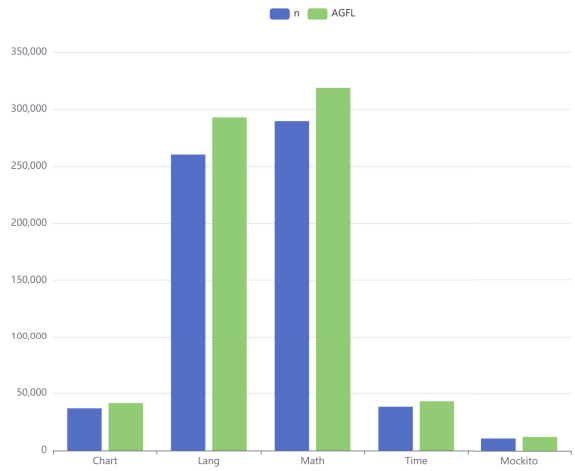

Fig. 3. Changes in the number of nodes after data processing

without data processing, and AGFL represents our proposed approach in this paper.

As shown in the TABLE III, GraphSMOTE improves the accuracy of fault localization greatly and significantly. Specifically, AGFL improves accuracy by 30%, 100%, 210%, 150%, 433% in each project, and in most cases only needed to examine one statement to identify the fault.

TABLE III
EFFECTIVENESS WITH OR WITHOUT SMOTE

Subjects	Techniques	Top-1	Top-3	Top5
Chart	n	10	10	10
	AGFL	14	15	15
Lang	n	21	24	24
	AGFL	42	46	46
Math	n	24	25	26
	AGFL	75	84	87
Time	n	6	7	7
	AGFL	15	15	15
Mockito	n	6	9	10
	AGFL	32	32	32

Summary for RQ2: The accuracy of AGFL is always higher than those techniques without SMOTE. Precisely, when examining the top 1 suspicious statement, the AGFL approach can localize more faults.

C. Answer for RQ3

An epoch is to complete work through all the training datasets. The learning rate (LR) controls model how quickly the training is adapted to the problem, determining the weight change generated in each cycle training process. The too-large epoch or too small learning rate will not improve the effect of the model in some cases but will lead to an over-fitting

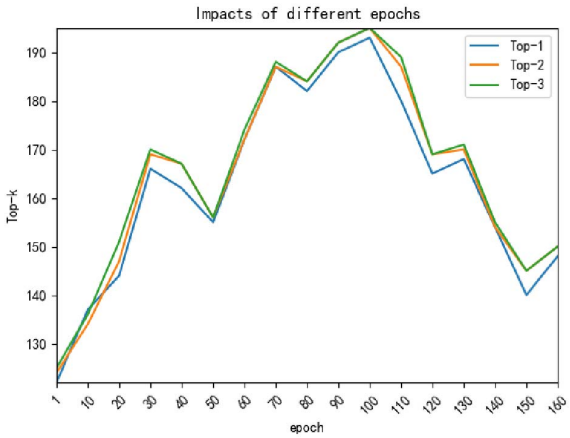

Fig. 4. Impacts of different epochs under lr=0.01

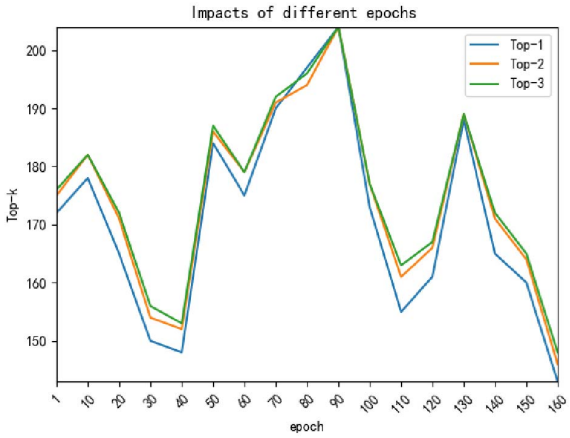

Fig. 5. Impacts of different epochs under lr=0.0001

phenomenon. In this paper, two different learning rates are considered to study the changes brought to the model by the increase of epoch. We set the learning rate at 0.01 and 0.0001, and the number of iterations is from 1 to 160. Fig. 4 shows the change of Top-k with the increase of epoch when the learning rate is 0.01. As can be seen from the figure, the value of Top-k increases as the number of epochs increases. However, when the epoch exceeded 120, Top-k values all declined, which we think is due to the over-fitting phenomenon caused by the increasing number of weight updates.

Fig. 5 indicates the change of Top-k with epoch when the learning rate is 0.0001. It can be seen that since epoch=1, LR=0.0001 has a better effect than LR =0.01. Similarly, the value of Top-k decreases after the number of epochs is more than 90. A Smaller learning rate gives better results than both. We consider that it is because when the learning rate is too large, the loss function may directly surpass the global optima, leading to the result being not optimal.

Summary for RQ3: A significant learning rate can make the model fast convergence, but the fault localization effectiveness is not the best. In summary, the default learning rate and epoch are the most stable and effective for AGFL in our experiment.

VI. Threats to Validity

Wohlin et al. [37] proposed that there are four basic types of validity threats that can affect the validity of an experiment. We focus on internal and external validity in the following paragraphs.

Threats to internal validity: The main threat to internal validity is the potential mistake in features collection and technique implementation. We collect features and implement our techniques by utilizing state-of-the-art tools and frameworks, such as Javalang, Word2Vec, and PyTorch, to reduce this threat.

Threats to external validity: To reduce these threats, we use five subjects from the Defects4J benchmark suite. Additionally, we use a graph neural network to train the model, resulting in different training results. To make the results more reliable, we calculate ten times and use the average as the result of the experimental study.

We encourage independent replications to assess the robustness of our results further.

VII. Related Work

Spectrum-based fault localization(SBFL) is one of the most famous fault localization approaches widely used in program debugging [38]. SBFL gathers the program entity information executed by the test case to identify the entity that is more likely to be buggy [39]. Jones et al. [5] proposed the Tarantula, which was the first to propose the use of spectrum to evaluate the program entity suspiciousness. For each program entity, Tarantula counts the number of execution of each program entity in all failed test cases and is divided by the number of execution both in all failed and passed test cases. Later, Wong et al. [7] proposed the Dstar formula to calculate the statement suspiciousness, believing that the statements covered by failed test cases have a high probability of faults. Experimental results show that Dstar is optimal compared with most SBFL approaches.

Mutation testing technology has made rapid progress with the enhancement of computer computing power. Mutation-based fault localization (MBFL) improves the accuracy of fault localization and can better deal with coincidentally correct test cases in SBFL. Papadakis et al. [11] used mutation tests for fault localization, mutation operation on the source code to generate mutants, and comparison of results after the execution of test cases to obtain suspiciousness. Moon et al. [12] use two sets of mutants. One set is the correct statement of mutants, and the other set is the wrong statement of mutants. Although MBFL improves the accuracy of fault localization, it consumes a considerable execution cost. Up to now, the study on MBFL mainly focuses on the reduction of mutation tests. Gong et al. [40] proposed the use of a dynamic mutation execution strategy to reduce the execution cost of MBFL. In addition, Papadakis et al. [41] reduce the number of mutants through random sampling of mutants, thus reducing the cost of MBFL.

Recently, many machine learning-based techniques are also applied to fault localization, mainly based on past information to predict where faults will occur. Wong et al. [42] implemented fault localization using BP neural networks with coverage information as input data and test case execution results as labels. Subsequently, BP neural network was used to calculate the suspiciousness of each executable statement. Liu et al. [14] propose an approach that uses graph-mining and support vector machines (SVM), where each node in the graph is an executed function. SVM is used to classify incorrect and correct executions. Meanwhile, with the advancement of deep learning, many techniques have been applied to software engineering. Zheng et al. [43] propose a fault localization method based on a deep neural network (DNN), which uses coverage data and test case results as input for training. Li et al. [34] used multi-layer perceptrons and treated suspiciousness value, fault proneness, and textual similarity as the characteristics of the model to achieve the location of faults.

VIII. Conclusion

The motivation of this study was to improve fault localization accuracy in real-world program debugging. To this end, we first proposed a method based on graph neural networks for single-fault locating efforts. The empirical study on 262 real bugs from the widely used Defects4J benchmark shows that AGFL can be an effective fault localization technique, e.g., Top-1 is better than other methods.

Our work is preliminary, but the result is encouraging and inspiring. In the future, there will be some aspects to improve and investigate. (1) We should introduce dynamic analysis of the code into the model features. (2) To shrink the graph structure without losing structural information, we need to reduce the number of nodes. (3) We still should take into account the efforts of patch generation. That is to say, bug fixing efforts also should be considered when building fault locating models.

Acknowledgements

This work is supported in part by the National Natural Science Foundation of China under Grant Nos. 61502497, 61673384, and 61673384; the Guangxi Key Laboratory of Trusted Software Research Plan under Grant KX201530 and Grant KX201532; and in part by the National Natural Science Foundation of Guangxi under Grant 2018GXNSFDA138003.

REFERENCES

[1] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok, "Recent catastrophic accidents: Investigating how software was responsible," in *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*. IEEE, 2010, pp. 14–22.

[2] W. E. Wong, X. Li, and P. A. Laplante, "Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures," *Journal of Systems and Software*, vol. 133, pp. 68–94, 2017.

[3] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[4] V. Debroy, W. E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," in *2010 10th International Conference on Quality Software*. IEEE, 2010, pp. 13–22.

[5] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[6] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[7] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 2012, pp. 21–30.

[8] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[9] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[10] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 765–784, 2013.

[11] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[12] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 153–162.

[13] D. Li, W. E. Wong, W. Wang, Y. Yao, and M. Chau, "Detection and mitigation of label-flipping attacks in federated learning systems with kpca and k-means," in *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2021, pp. 551–559.

[14] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu, "Mining behavior graphs for "backtrace" of noncrashing bugs," in *Proceedings of the 2005 SIAM international conference on data mining*. SIAM, 2005, pp. 286–297.

[15] W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an rbf neural network to locate program bugs," in *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2008, pp. 27–36.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[17] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[18] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[19] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 368–377.

[20] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[21] S. Paul and A. Prakash, "A framework for source code search using program patterns," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463–475, 1994.

[22] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[23] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[24] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[25] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[26] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[28] N. V. Chawla, N. Japkowicz, and A. Kotcz, "Special issue on learning from imbalanced data sets," *ACM SIGKDD explorations newsletter*, vol. 6, no. 1, pp. 1–6, 2004.

[29] R. Mohammed, J. Rawashdeh, and M. Abdullah, "Machine learning with oversampling and undersampling techniques: overview study and experimental results," in *2020 11th International Conference on Information and Communication Systems (ICICS)*. IEEE, 2020, pp. 243–248.

[30] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[31] T. Zhao, X. Zhang, and S. Wang, "Graphsmote: Imbalanced node classification on graphs with graph neural networks," in *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, 2021, pp. 833–841.

[32] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 184–195.

[33] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 165–176.

[34] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[35] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 273–283.

[36] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[37] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[38] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu, "Spectrum-based fault localization: Testing oracles are no longer mandatory," in *2011 11th International Conference on Quality Software*. IEEE, 2011, pp. 1–10.

[39] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *arXiv preprint arXiv:1607.04347*, 2016.

[40] P. Gong, R. Zhao, and Z. Li, "Faster mutation-based fault localization with a novel mutation execution strategy," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–10.

[41] M. Papadakis and Y. Le Traon, "Effective fault localization via mutation analysis: A selective mutation approach," in *Proceedings of the 29th annual ACM symposium on applied computing*, 2014, pp. 1293–1300.

[42] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 04, pp. 573–597, 2009.

[43] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," *Mathematical Problems in Engineering*, vol. 2016, 2016.