Automated Software Engineering (2023) 30:16
https://doi.org/10.1007/s10515-023-00383-z

# GNet4FL: effective fault localization via graph convolutional neural network

**Jie Qian**[1] · **Xiaolin Ju**[1] · **Xiang Chen**[1]

Received: 26 August 2022 / Accepted: 9 April 2023


**Abstract**
Fault localization aims to efficiently locate faults when debugging programs, reducing software development and maintenance costs. Spectrum-based fault location (SBFL) is the most commonly used fault location technology, which calculates and ranks the suspicious value of each program entity with a specific formula by counting the coverage information of all the program entities and execution results of test cases. However, previous SBFL techniques suffered from low accuracy due to the sole use of execution coverage. This paper proposed an approach GNet4FL based on the graph convolutional neural network. GNet4FL first collects static features based on code structure and dynamic features based on test results. Then, GNet4FL uses GraphSAGE to obtain node representation of source codes and performs feature fusion on an entity consisting of multiple nodes, which preserves the topological information of the graph. Finally, the representation of each entity is input to the multi-layer perceptron for training and ranking entities. The results of the study showed that GNet4FL successfully located 160 out of 262 faults, outperforming the three state-of-the-art methods by 94, 42, and 14% in Top-1 accuracy, and having close results to Grace with less cost. Furthermore, we investigated the impact of each component (i.e., graph neural network, pruning, and dynamic features) of GNet4FL on the results. We found that all of these components had a positive impact on the proposed approach.

**Keywords** Fault localization · Graph convolutional neural network · Abstract syntax tree

Xiaolin Ju and Xiang Chen have contributed equally to this work.

✉ Xiaolin Ju
ju.xl@ntu.edu.cn

Extended author information available on the last page of the article

Published online: 24 April 2023

## 1 Introduction

Program faults are inevitable during software development and evolution (Xuan et al. 2014). Developers must spend extensive time on testing and debugging to improve software quantity. Moreover, testing and debugging of a program can account for more than 75% of its project budget (Planning 2002). To improve the efficiency of program debugging, researchers proposed different automatic fault localization techniques, which generate fault localization reports with minimal human intervention to assist developers in debugging (Wong et al. 2016). Traditional approaches for locating faults always rely on the execution information of programs. Spectrum-based software fault localization (SBFL) designs ranking metrics to calculate statements' suspiciousness by analyzing the coverage information and execution results of test cases (Abreu et al. 2009). An early influential study (named Tarantula) was proposed by Jones and Harrold (2005), which figured out the basic heuristic of SBFL that entities covered by the failed test cases are more likely to be faulty than those covered by the passed test cases. Later, various automatic fault localization approaches were proposed by designing new risk formulas or modifying the existing coefficients, such as DStar, SOBER, Falcon (Wong et al. 2013; Liu et al. 2005; Park et al. 2010; Wong et al. 2010). However, SBFL approaches only consider the program coverage information and execution result. Therefore it is hard to achieve high accuracy without enough coverage information or due to the presence of coincidental correctness (Feyzi 2020).

Recently, several techniques for fault localization, called Mutation-based fault localization (MBFL), were proposed to improve fault localization accuracy. These MBFL techniques can analyze different program behaviors by mutating code entities with simple syntactic code changes, such as replacing $if(a > 0)$ with $if(a < 0)$ (Pearson et al. 2017). Although MBFL can obtain enough coverage information about the execution of statements in different situations, it is less efficient due to the vast computation cost. Zou et al. (2019) evaluated the effectiveness of SBFL, MBFL, and other methods on Defects4J. They claimed that SBFL is the most valuable independent fault localization method.

Deep learning techniques have made significant achievements in current software engineering studies, such as AI for software engineering and software engineering for AI (Yang et al. 2021; Zhang et al. 2020). Various fault localization methods based on deep learning technologies have been proposed. These methods extract various features of software under testing and evaluate suspiciousness scores based on the basic idea of SBFL or MBFL. Then, they are input into the deep learning model for training to obtain the ranking list. For example, DeepFL (Li et al. 2019) applied multi-layer perceptron (MLP) to analyze suspiciousness scores based on SBFL and MBFL, code complexity, and text similarity, respectively. FLUCCS (Sohn and Yoo 2017) learned to rank program entities based on suspiciousness scores of existing SBFLs, code complexity, and code history.

Moreover, static fault localization techniques identify fault by analyzing the code structures and inferring the program semantics. Abstract syntax tree (AST), which can represent the syntactic structure of source code, is widely used in static fault localization techniques (Qian et al. 2021). Furthermore, AST-based neural models can better represent the source code (Zhang et al. 2019a). Graph embedding learns to represent each node as a vector by aggregating and structuring information. Compared with word embedding, it improves the performance of computing and provides a new way to consider semantics and structural relationships (Gu et al. 2021).

Graph neural networks can better consider the program's context and perform better (Wu et al. 2020). This study proposes an AST-based fault localization method via a graph neural network called GraphSAGE, which uses sampling techniques to overcome the memory limitations of traditional graph neural network training (Hamilton et al. 2017). This paper differs from our previous work (Qian et al. 2021) by adding coverage information, reducing the size of the graph structure, and outputting the values for each program entity via MLP. It first learns the features of each node based on the AST structure and coverage information, then utilizing MLP to calculate the statements' suspiciousness. To evaluate the effectiveness of the proposed method, we also conducted an empirical study on five projects from the widely used Defects4J benchmark. Our empirical study indicates that GNet4FL can successfully locate 160 faults out of 262 in the Top-1, which is significantly better than the state-of-the-art baselines [Ochiai Abreu et al. (2006), FLUCCS Sohn and Yoo (2017), DeepFL Li et al. (2019)]. Furthermore, we investigate the impact on the performance of each component of GNet4FL.

In summary, our paper makes the following contributions:

- GNet4FL: A fault localization method based on graph convolutional neural networks to predict potential faulty entities via a combination of static and dynamic features of source code. GNet4FL combined coverage information and source code representation as features. In particular, the GraphSAGE aggregates neighboring node embedding from the entire graph structure. MLP is applied to rank the program entities.
- Empirical Study: We empirically evaluated GNet4FL on five projects from Defects4J. The results demonstrate the effectiveness of GNet4FL in locating real-world faults. The source code of GNet4FL has been made available at the following website.[1]

The rest of this paper is organized as follows. Section 2 describes the background of this research topic. Section 3 presents the proposed model with detailed description. Section 4 describes the design of research questions and experimental methods, evaluation metrics. Section 5 evaluates the performance of GNet4FL, including result comparison and analysis. Section 6 discussed the threats to the validity of our work. Finally, we summarized the paper in the last section.

[1] https://github.com/humorrr/GNet4FL.

## 2 Background and related works

In this section, we first summarized the state of the art in fault localization and emphasized the novelty of our work. Then we introduce the principles of graph convolutional neural networks. The goal of our work aims to learn source code context information via graph convolutional neural networks to obtain AST node representations.

### 2.1 Related works

There are various fault localization techniques proposed in the past decades, such as spectrum-based fault localization, mutation-base fault localization, Information Retrieval (IR) based fault localization, Deep learning-based fault localization, etc. This section will summarize these above techniques as follow.

Spectrum-based fault localization (SBFL) is one of the most used and classic technique. It is mainly based on the results of the test cases and whether the statements are covered or not to design the suspiciousness formula. Several ranking formulas exist, including Tarantula (Jones and Harrold 2005), DStar (Wong et al. 2013), Naish (Naish et al. 2011), Ochiai (Abreu et al. 2006), and others. Xie et al. (2013a) conducted a theoretical study on the validity of each formula and proved two groups of optimal formulas. Furthermore, SBFL is often combined with other techniques (such as program slice, function call graph, etc.) to improve accuracy (Ju et al. 2014; Vancsics et al. 2022; Mao et al. 2014; Xie et al. 2013b; Soremekun et al. 2021). Zhang et al. (2019b) first analyzed the importance of different test cases and applied PageRank to compute the spectral information.

Mutation-Based Fault Localization (MBFL) means executing a test case on a modified program and then calculating the suspiciousness value of the program entity based on the results (Dutta and Godboley 2021). The rules that modify the program are called mutation operators, and the program is called after the execution of the mutation operators. Papadakis and Le Traon (2015) use the execution information before and after the mutation to calculate the suspiciousness value. When the results differ, the mutant is said to be killed; conversely, it is not killed. The formula in SBFL [e.g., Ochiai Abreu et al. (2006)] is then consulted to calculate the suspiciousness of the program entity. However, it is inefficient due to the long time required to generate mutants, which is a drawback of MBFL. Liu et al. (2017) selected mutants proportionally and statements covered by failed test cases multiple times to reduce the number of mutants generated. Their approach reduces the execution cost of MBFL.

IR-Based Fault Localization (IRBL) selects the program entity related to the fault report according to the similarity between the report and the program. IRBL is designed to get suspicious program modules simply by typing fault reports. Zhou et al. (2012) proposed the fault locator method based on code similarity and the revised Vector Space Model (rVSM) to locate faulty files.

Based on the above approach, researchers considered combining multiple information. Peng et al. (2020) combined source code features and 14 types of SBFL

scores to rank program statements by encoder learning features and SVMRank. Xiao et al. (2021) used code semantics, 11 types of SBFL, and three types of MBFL method scores as features, followed by an attention mechanism and LambdaRank to obtain a list of FL reports. Li et al. (2019) used SBFL, MBFL, and IRBL information as features and implemented fault localization via MLP.

Deep learning methods became popular for fault localization recently. Wong and Qi (2009) proposed using program spectrum information to train a BP neural network and then calculate the suspiciousness of each utterance. Lam et al. (2017) proposed DNNLOC by combining deep learning and information retrieval. It uses the rVSM to generate fault reports with source code similarity features and then uses Deep Neural Networks (DNN) to associate the fault reports with the source code. Zhang et al. (2021) investigated the effectiveness of three deep learning algorithms (CNN, RNN, and MLP) for localizing faults in eight real-world programs.

Recently, researchers have applied traditional deep learning to graph structures (Qian et al. 2021). This aims to learn via the structural information and attributes of graphs. Due to their high ability to process graph data, graph neural networks have been widely used in areas of social networks, recommendation systems, link prediction, and other fields. Xu et al. (2020) built graph neural network models to predict software defects, which use community detection methods to prune ASTs. The graph neural network is then used to capture source code contextual information. In fault localization, Lou et al. (2021) first applied graph neural networks to learn graph representations and obtained a ranked list using the learning-to-rank algorithm. This method uses test cases as nodes in the graph and constructs edges between test cases and code nodes, which can effectively provide dynamic information to the code nodes, but this results in a more massive graph. Therefore, this paper uses the results of the test cases directly as node features and fuses the multi-node to to reduce the graph size as much as possible while retaining good performance. Moreover, a graph-level representation is used to compute probabilities via MLP.

### 2.2 Graph convolution neural network

The primary idea of a Graph convolution neural network (GCN) is to learn representation for nodes by smoothing features over the graph (Kipf and Welling 2017; Wu et al. 2019). GCN can be divided into two categories: Spectral and Spatial. Bruna et al. (2013) proposed the first spectral graph convolution neural network in 2013. They defined the graph convolution in spectral space based on graph and convolution theorem. Traditional deep learning methods mainly use batch size to solve large-scale training data problems. GCN utilizes graph convolution to obtain the embeddings of all adjacent nodes of each node. During back-propagation, all the embeddings in the graph are stored in memory, which requires a large amount of memory for the experimental device to process the whole graph data. Meanwhile, GCN is a Transductive learning method. i.e., Transductive learning uses training and test data to train the model and then uses test data again to test the accuracy.

GraphSAGE samples neighbor nodes randomly so that each node's adjacent nodes are less than a given number of samples (Hamilton et al. 2017). Taking a node as the target node, GraphSAGE firstly randomly sampled its first-order and second-order neighbors and only considered the sampled node as the relevant node. Then, the model applied the aggregation function to the relevant node's features and updated the node's feature representation with the aggregation results. GraphSAGE proposes three aggregation functions: *Pooling Aggregator*, *Mean Aggregator*, and *LSTM*. Pooling and Mean Aggregator refer to relevant nodes' maximum and means values as aggregation results; LSTM refers to the input of related nodes into the model and the output as an aggregation result. Considering the running time, we use Mean Aggregator (computed as Eq. 1) as the aggregation function.

$$\mathbf{h}_v^k \leftarrow \sigma\big(\mathbf{W} \cdot \mathrm{MEAN}\big(\big\{\mathbf{h}_v^{k-1}\big\} \cup \big\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\big\}\big)\big). \tag{1}$$

where W is the weight matrix of each layer to be learned by training, $k$ represents the $k$th layer, $h_v^k$ denotes the node embedding of node $v$ after the $k$th aggregations, $\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)$ denotes the $(k-1)$th layer vector of neighboring embedding, and $\sigma$ denotes the nonlinear transformation. The $(k-1)$th layer vector of node $v$ vertices and neighboring vertices are concatenated and averaged, and the result is transformed non-linearly to produce the $k$th layer embedding of node $v$.

GraphSAGE and GCN apply feature aggregation on adjacent nodes. The main difference is that GraphSAGE's aggregation is more flexible than that of GCN. GraphSAGE is inductive learning, which uses a trained model to predict test data and reduce the number of node features for training. When new nodes are added to the graph, the aggregation method can make GraphSAGE get the latest representation without iterate training. This operation prevents the over-fitting of model training effectively and enhances generalization ability. GraphSAGE uses a particular sampling method to solve the memory problem, suitable for large-scale graphs.

## 3 Our approach

The framework of our approach can be found in Fig. 1. Our approach features the use of GraphSAGE and MLP to evaluate the risk of each statement. As shown in Fig. 1, GNet4FL is divided into two stages including `Data Preprocessing` and `Model Construction`. The first stage computes the AST and extracts features of examined source code. The second stage constructs the prediction model after GraphSAGE and applies MLP to sort statements. Next, we will introduce the implementation of these two stages in sequence.

### 3.1 Data preprocessing

Abstract syntax code (AST), the tree sketch of a source code, represents the syntactic structure of source code semantics and information. Compared to the source code, an AST does not contain inessential delimiters and punctuation (such as

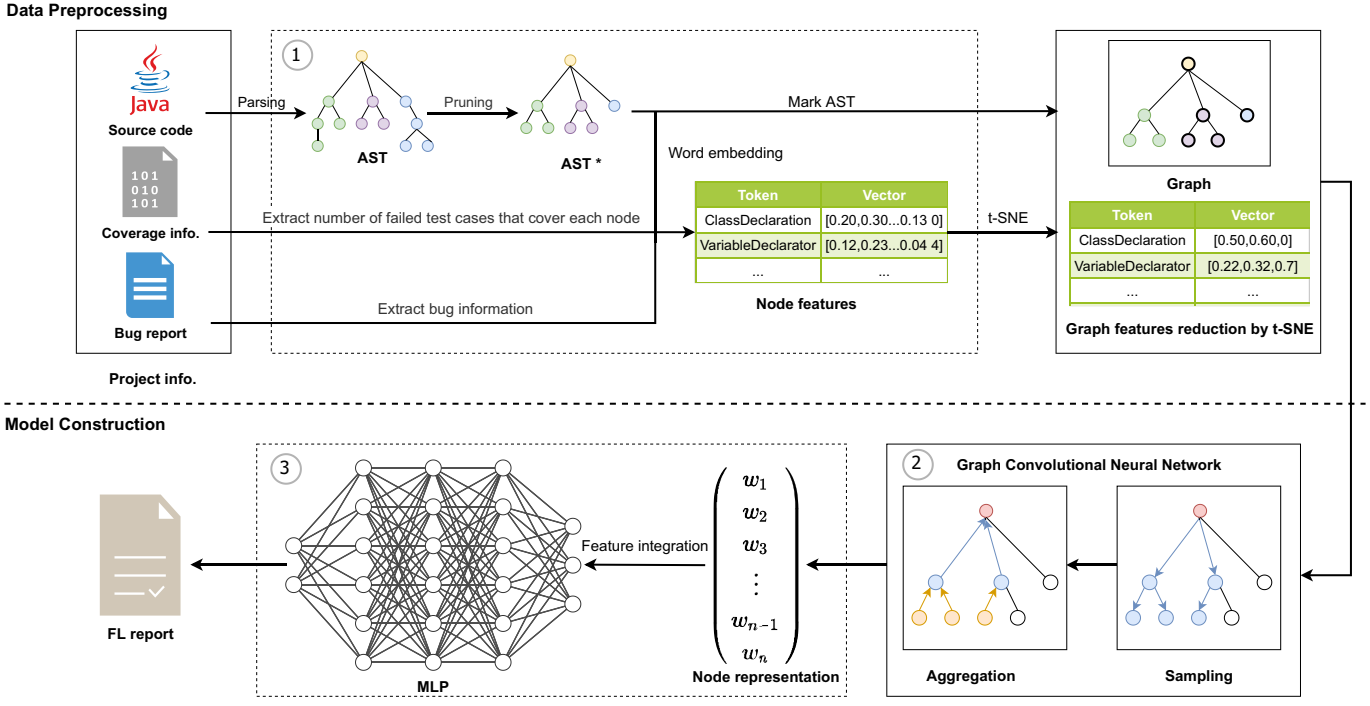


**Fig. 1** Framework of our approach

braces, semicolons, parentheses, etc.). Furthermore, an AST usually contains extra details about the program due to the consecutive stages of analysis by the compiler. Many researchers applied AST to examine source code attributes for successive software maintenance activities. For example, Zhang et al. (2019a) proposed the ASTNN method based on AST for code representation. Alon et al. (2019) proposed the model of Code2Vec, which parses code into AST and extracts different paths to implement code representation.

Firstly, we tokenized the source programs into ASTs with Javalang. In constructing the AST, Javalang divides the nodes into multiple types. However, this will increase the size of the graph structure of ASTs. Therefore, we pruned the original ASTs to remove the unnecessary nodes. Generally speaking, refining the statements to the token level will have more redundant nodes and thus affect the classification results. Besides, Zhang et al. (2019a) claimed that the token-based method caught little information. When redundant information is applied to the sample, it will significantly increase the proportion of correct and wrong nodes and be detrimental to the model's prediction. If we use statement-level ASTs, some critical information will be lost (such as while and switch statements). Compared to the token-level graph, it has half or even fewer nodes and edges. Therefore, it is necessary to combine these two levels of ASTs.

Considering that the granularity of the statement is not a reasonable representation of the structure information for control-flow constructs (for example, Fig. 2a), we mainly aim to process control-flow nodes (e.g., if and for statement). For other types of nodes (e.g., function and variable declaration), we use the name of the node corresponding in Javalang as the node name, i.e., the parent node of each statement. This paper presents a pruning algorithm that combines two granularities, token and statement. To do this, the Javalang tool is used to first obtain a token-level AST. Next, the MethodDeclaration node is identified in each AST, dividing it into smaller parts. Finally, the non-loop statement nodes are removed and the parent nodes representing the statement are kept, thereby completing the pruning process.

Taking the example of code in Mockito 33, we start with each line of statements using the parent node type as the node name, as illustrated in Fig. 2a. Second, refine the for loop to the token level. In the **ForStatement** node, there are two types of child nodes: the **ForControl** node and the **BlockStatement** node. Considering that there is a high probability of conditional faults in loop statements, we continue parsing for conditional statements, as illustrated in Fig. 2b.

In the first step, we tokenized and transformed the source code into multiple ASTs to obtain syntactic structure information. Then, the ASTs are pruned to map statements in the source code to one or more nodes, and the undirected graph $\mathcal{G}=(\mathcal{V},\mathcal{E})$, which consists of a set of node $\mathcal{V}$ and edges $\mathcal{E}$, are constructed. Finally, the adjacency matrix $A \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{V}|}$ that can represent the graph information is obtained by traversing the graph, where A[i][j] = 1 indicates that node $i$ is connected with $j$. Additionally, attributes for each node represent the node name, e.g., ClassDeclaration, WhileStatement. To convert words into numeric form, we apply Word2vec to word embedding. Word embedding maps words to generate a low-dimensional vector representation in a new space, ranging from dozens to thousands of dimensions. In word embedding, we set the word vector dimension

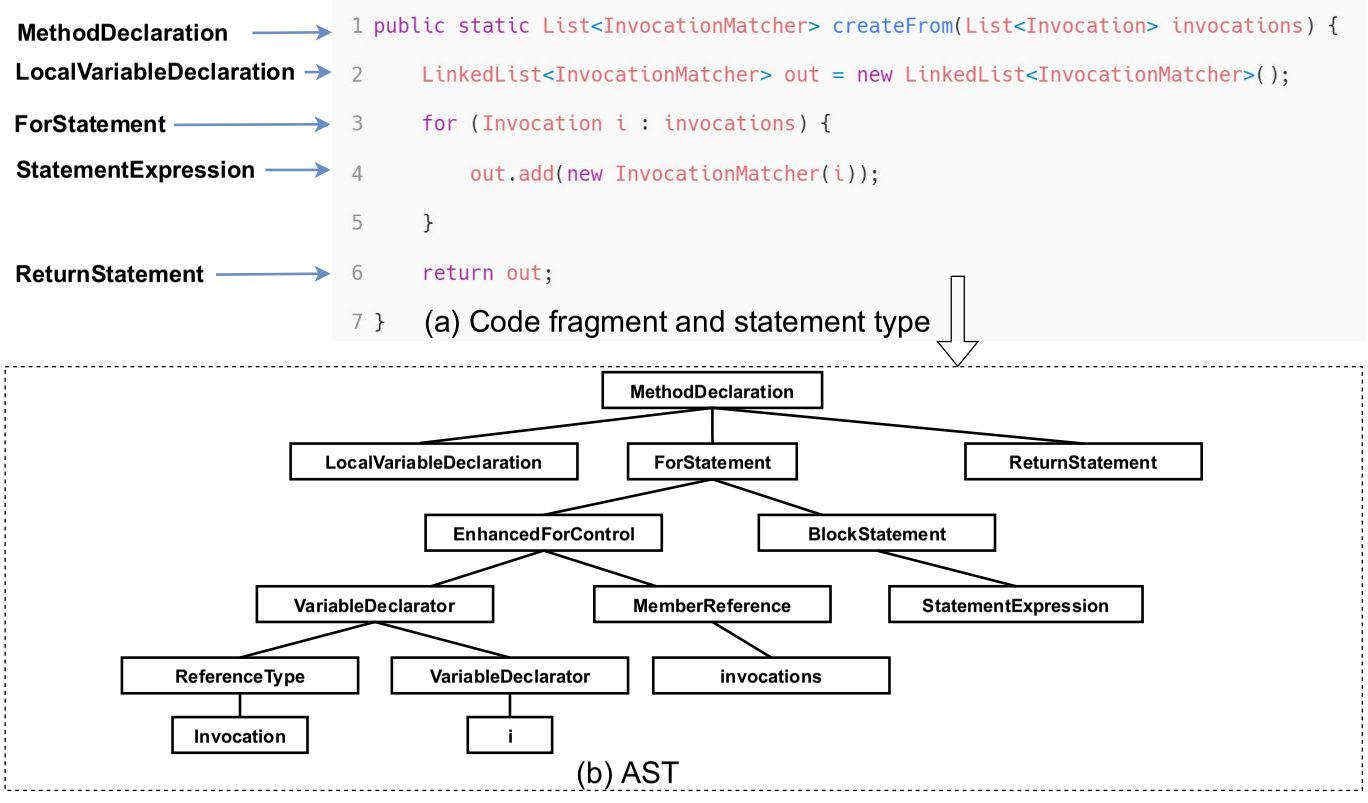


**Fig. 2** An example of source code with corresponding abstract syntax tree (AST)

to 256. However, higher-dimensional vectors do not necessarily work well in GNN. Meanwhile, it will also have higher requirements on the memory of the running device in the next stage. If we directly set the word vector's dimension to a small size using Word2vec, the words will be clustered together. Raunak (2017) used a dimensionality reduction method based on PCA to construct low-dimensional word embedding. They discuss that the dimensionally reduced vector can maintain similar or better results than the original feature. Anowar et al. (2021) compared several feature extraction algorithms and analyzed their effectiveness in classification problems. In the binary classification problem, the 2-dimensional T-distributed neighbor embedding (T-SNE) features were only 0.0484 lower than the 200-dimensional features of the best KPCA in the F1-score metric. Secondly, the non-linear methods outperformed the linear ones in most cases. And other methods will require the selection of a suitable dimension for dimensionality reduction, which adds a parameter to the model and thus increases the randomness of the final result.

Inspired by the work of Anowar et al. (2021), we apply T-SNE to reduce the dimension for the original embeddings. Van der Maaten and Hinton (2008) propose a machine learning algorithm for dimensionality reduction called T-SNE. The main idea is to convert the euclidean distance into conditional probability to represent the similarity between points. Then, the gradient descent algorithm is applied to fit the high-dimensional distribution after the obtained word vector is used as the feature of the node $X \in \mathbb{R}^{|\mathcal{V}| \times d}$, where $d$ represents the dimension of the feature vector.

Fault localization techniques can be divided into two categories (static and dynamic) depending on whether test cases need to be executed. The method based on AST belongs to static categories by determining the location of faults with both the code structure and syntax information of source files. The dynamic method analyzes the behavior of programs under the test by executing a set of test cases. Considering that the statements covered by failed test cases are more likely to have larger suspicious scores (Wong et al. 2013), we introduce coverage information of AST nodes together with the execution results as the features of the programs under test. Specifically, we count the number of failed test cases that cover each AST node. If failed test cases do not cover the node, the number of times is set to 0. Otherwise, the number is assigned to $n$ ($n$ is the times of the AST node covered by the failed test cases). Then, the AST node coverage times obtained are combined with node feature $X$ to form a new feature $\tilde{X} \in \mathbb{R}^{|\mathcal{V}| \times d+1}$. Bug reports are used to identify the locations of faulty lines and mark nodes to differentiate between correct and faulty nodes. This step is intended to obtain the locations of faults in the training data for the supervised learning period of our model.

### 3.2 Model construction

Data Preprocessing, extracting the statement information of the source code under test, is to transform the source program into a graph (AST) and node features as the input of our model. As shown in step 2 of Fig. 1, GraphSAGE transforms the feature

representation of nodes by sampling and aggregating and uses the Relu activation function to realize non-linear transformation.

After step 2 (GraphSAGE), we obtain the node representation for each AST node. Due to our operations on the AST, a single statement is represented by one or more nodes. Wang et al. (2019) proposed the Heterogeneous graph attention network (HAN), which combines different semantic information. Illuminated by the work of Wang et al. (2019), we refer to the treatment of multi-type features of nodes in heterogeneous graphs when there are multiple representations of AST nodes.

Specifically, we compare the node representation to semantic information in HAN. Suppose that the node $Q$ has $k$ nodes representing $\mathbf{z}_i$. Then the node $Q$ is aggregated in the form of a sum of weights (Eq. 2).

$$F = \frac{1}{k} \sum_{i \in k} \cdot \tanh\left(\mathbf{W} \cdot \mathbf{z}_i + \mathbf{b}\right) \tag{2}$$

where $F$ represents the final node representation of node $Q$, $\mathbf{z}_i$ is the feature of each node, $W$ is the weight matrix of the one-layer neural network, and $b$ is the bias.

The $k$ nodes of the statement are linearly transformed by *tanh*-function method, and the mean value is taken as its feature. Next, we input the resulting data directly into the MLP. Since we regard fault localization as a classification problem, the softmax function is used to output the probability of each statement as shown in Eq. (3),

$$\text{Softmax}\left(y_i'\right) = \frac{e^{y_i'}}{\sum_{j=1}^{n} e^{y_j'}} \tag{3}$$

where $y_i'$ is the output of node $i$ and $n$ is the number of all nodes. The softmax function can convert the output values to probabilities in the range [0, 1]. Finally, the output is treated as the suspiciousness score of the statement and sorted to generate an fault localization (FL) report.

## 4 Empirical study

### 4.1 Research questions

In our empirical study, we want to evaluate the performance of our framework (GNet4FL) by answering the following four research questions (RQs)

**RQ1**: Compared with state-of-the-art fault localization techniques, how does GNet4FL perform in locating real-world faults?

**Motivation.** In this RQ, we wanted to compare the performance of GNet4FL with four baselines. We have chosen a classical SBFL method (Ochiai Abreu et al. 2006) and three state-of-the-art deep learning methods (FLUCCS Sohn and Yoo 2017, DeepFL Li et al. 2019 and GRACE Lou et al. 2021) as the compared baselines.

**Table 1** Characteristics of subjects

| Subject | Project name | LOC(k) | #Fault |
|---|---|---|---|
| Chart | jfreechart | 96 | 26 |
| Time | joda-time | 28 | 27 |
| Mockito | mockito | 23 | 38 |
| Lang | commons-lang | 22 | 65 |
| Math | commons-math | 85 | 106 |
| Total | | 254 | 262 |

**RQ2**: Do graph neural networks improve the performance over traditional deep learning-based fault localization methods?

**Motivation.** In this RQ, we wanted to investigate the impact of performance by applying graph neural networks on GNet4FL. To answer this research question, we compare it with three machine learning methods.

**RQ3**: How influential are the main components of effectiveness in locating faults, such as pruning and dynamic features, serve of GNet4FL?

**Motivation.** In this RQ, we wanted to explore the effectiveness of combining various methods and the impact of each component on the results. To answer this question, we compared the effects of each component before and after its application.

**RQ4**: How does GNet4FL perform in cross-project fault localization?

**Motivation.** In RQ1, RQ2, and RQ3, we mainly evaluated the performance of GNet4FL on the same project. Then In RQ4, we aim to investigate the generalization of our proposed method. That is to say, we trained our model with some subjects then test the performance with another subject. This is because the training history data is usually not the same as the data from the examined program.

### 4.2 Experimental setup

We applied GNet4FL on the benchmark (Defects4J V1.2.0[2]) which has been widely used on most fault localization researches (Küçük et al. 2021; Gong et al. 2015; Lou et al. 2021; Qian et al. 2021) to evaluate the performances of our approach. Defects4J includes six real-world projects with 395 faults. Table 1 illustrated the characteristics of these five subjects examined in our empirical study.

Furthermore, we adopted an objective solution under the following situations to better evaluate the performance of fault localization on real-world faults:

1. **suspiciousness score**: We can calculate the fault localization ranking lists with the prediction model, i.e., each statement corresponds to a suspiciousness score. When multiple statements score the same as the faulty element, we average the best and worst cases for these elements (Keller et al. 2017). The best case is when the faulty statement is the first to be checked among all statements with the

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

same suspiciousness. The worst case is when the faulty statement is last checked (Wong et al. 2016). For example, if two statements have the same score as the faulty statement and are ranked $i$ in the list, these statements are all considered to be ranked $i$+1.

2. **faults of omission**: In some programs, failures are not caused by incorrect assignment statements or predicates but by missing code. According to our statistics, Lang and Math in Defects4J have more than one omission fault, accounting for 40% and 30%, respectively. Inspired by previous work (Pearson et al. 2017; Küçük et al. 2021), we manually examined a set of candidate positions, typically mostly the start and end positions of a code block. The predicted line is considered successful when it is between a code block's start and end positions.
3. **multiple faults**: Extensive programs in the real world mostly contain numerous faults. In this paper, we consider that software only contains a single fault. Furthermore, one type of fault is a single fault that spans various statements. We assume success when checking any one of these multiple statements.

We apply Word2vec[3] with the Skip-Gram algorithm (Chung et al. 2002) to obtain word vectors and with the Javalang tool[4] to generate AST. In the embedding layer, the dimension of Word2Vec for the initial node representation is 256. In the graph hidden layer, we set the dimension of hidden states as 20. The hyper-parameters of our model are tuned with the Microsoft NNI tool,[5] and then determine the optimal parameters of our model. The $k$-fold cross-validation is applied to our research due to its high accuracy (Fushiki 2011). After performing both ten-fold and five-fold cross-validation on Mockito to evaluate their effectiveness and efficiency, it was found that five-fold cross-validation was equally effective with less time-consuming. As a result, we adopted five-fold cross-validation in our empirical studies. We build our model relying on PyTorch V1.60.[6] In the first three research questions, we used full-batch for our experiments. In RQ4, we use PyTorch-Geometric's mini-batch and set the batch size to 512 to minimize the cost of model training. Due to equipment limitations, we used Grace's results directly but kept them as close to its experimental settings as possible for a fair comparison.

### 4.3 Evaluation metrics

This paper evaluates the performance of GNet4FL by Top-$k$ and *MAR*, which are also widely used in existing studies (Li and Zhang 2017; Li et al. 2021).

[3] https://code.google.com/p/word2vec.
[4] https://github.com/c2nes/javalang.
[5] https://github.com/microsoft/nni.
[6] https://pytorch.org.

**Table 2** Graph structure information and extracting time costs of each subject

| Subject | #vertex | #edges | Time (s) |
|---|---|---|---|
| Chart | 29,698 | 29,672 | 3.28 |
| Time | 29,014 | 28,988 | 2.84 |
| Mockito | 7070 | 7032 | 0.65 |
| Lang | 85,574 | 85,510 | 8.83 |
| Math | 74,740 | 74,634 | 10.2 |

#### 4.3.1 Recall at Top-k

Recall at Top-$k$ means the number of program faults in Top-$k$ entities in the rank list, which indicates the performance of the rank list generated by a fault localization method. Following the recommendations of Li et al. (2019, 2021) and Lou et al. (2021) we set $k = 1, 3, 5$.

#### 4.3.2 Mean average rank

The mean average rank (MAR) for a set of faults is the mean of the average rank scores for each fault.

$$\text{MAR} = \frac{\sum_{b=1}^{B} AveR(b)}{B} \tag{4}$$

where $B$ is faults count, $AveR(b)$ is the average rank of fault $b$. MAR aims to evaluate the average ranking of the faults within each project.

## 5 Result analysis

This section presents the answer to four research questions (RQs) raised in Sect. 4. We first establish the analysis procedures to answer these RQs. Next, we introduce and summarize the experimental results.

### 5.1 Result analysis for RQ1

To effectively evaluate the performance of GNet4FL for fault localization, the following four approaches have been selected as baselines for comparison:

1. Ochiai (Abreu et al. 2006): It is a classic spectrum-based fault localization method, often used as one of the baselines for comparison.

2. FLUCCS (Sohn and Yoo 2017): It assembles ranking matrices of multiple methods as features, then applies genetic algorithms to rank suspiciousness of program entities based on support vector machines (SVM).
3. DeepFL (Li et al. 2019): It applies MLP to consider various feature dimensions (including suspiciousness scores based on SBFL and MBFL, code complexity, and text similarity).
4. GRACE (Lou et al. 2021): It is based on coverage and graph representation learning. To the best of our knowledge, GRACE is the first to apply graph neural networks in fault localization.

Table 2 describes the graph structure of five projects, including the number of nodes, edges, and the time of constructing the graph representation for each project. As Table 2 shows, the time cost for constructing the graph for each project is less than 11 s. Furthermore, considering the LOC column in Table 1, we observe that in project `Time`, the number of nodes is inversely proportional to the lines of code compared with other projects, such as `Chart`, `Lang`, and `Math`. This is due to the presence of looping structures that result in finer granularity and higher number of nodes in the ASTs. Additionally, the data preprocessing (including word embedding and dimensionality reduction) took 210–261 s.

Table 3 presents the effectiveness measurements by applying the four above approaches on the dataset listed in Sect. 4. The first column is the name of projects in Defects4J, the second column is the name of the fault localization techniques, and the remaining columns are the results for the metrics Top-1, Top-3, Top-5, and MAR. The results show that GNet4FL outperforms or comes close to compared techniques, successfully locating 160, 204, and 217 faults of the 262 faulty programs by Top-1, Top-3, and Top-5. Specifically, GNet4FL can locate faults 94 more than Ochiai in Top-1, 42 more than FLUCCS, 14 more than DeepFL, and 12 more than GRACE, as well as a maximum improvement of 58.75%. To better evaluate the effectiveness of GNet4FL against Grace, this paper also uses Grace's setting (i.e. using leave-one-out cross-validation and worst ranking). The results show that GNet4FL locates 145, 188, and 208 faults within Top-1, Top-3, and Top-5, with results close to the performance of Grace. Moreover, this paper compares with another deep learning-based approach, DeepRL4FL, which proposes an enhanced coverage matrix and uses fully connected layers to improve performance (Li et al. 2021). Unfortunately, as their project is not open-sourced and challenging to reproduce, the results in this paper are cited directly from their paper. DeepRL4FL achieved Top-1, Top-3, and Top-5 scores of 71, 128, and 142 on six projects respectively, all of which is lower than those of GNet4FL on five projects. Furthermore, GNet4FL had the best MAR in 3 out of five projects. We employed the Wilcoxon rank-sum test to calculate the $p$ value for detecting the differences between GNet4FL and its baselines, then corrected the $p$ values using the Benjamini-Hochberg (BH) method (Yang et al. 2016). Wilcoxon signed-rank test is a kind of sign test in nonparametric statistics, which considers not only the positive or negative difference between the observed value and the center position of the null hypothesis but also the magnitude of that difference (Woolson 2007). We set the significance level at 0.05, indicating that if the corrected $p$ values are below this value, there is a

**Table 3** Effectiveness of GNet4FL and compared techniques

| Subject | Technique | Top-1 | Top-3 | Top5 | MAR | Training(s) |
|---|---|---|---|---|---|---|
| Chart | Ochiai | 6 | 14 | 15 | 9 | – |
| | FLUCCS | 15 | 19 | 20 | 8.08 | 12.33 |
| | DeepFL | 12 | 20 | 20 | 5.52 | 10.07 |
| | GRACE | 16 | 20 | 22 | 3.84 | – |
| | GNet4FL | 19 | 22 | 23 | 3.62 | 61.28 |
| Time | Ochiai | 6 | 11 | 13 | 15.96 | – |
| | FLUCCS | 8 | 15 | 18 | 9 | 9.52 |
| | DeepFL | 13 | 17 | 17 | 12.68 | 6.87 |
| | GRACE | 11 | 16 | 20 | 8.8 | – |
| | GNet4FL | 14 | 18 | 20 | 9.52 | 43.27 |
| Mockito | Ochiai | 7 | 14 | 18 | 20.22 | – |
| | FLUCCS | 7 | 19 | 22 | 14.78 | 12.48 |
| | DeepFL | 12 | 19 | 22 | 13.42 | 10.51 |
| | GRACE | 15 | 22 | 26 | 8.06 | – |
| | GNet4FL | 18 | 27 | 29 | 8.42 | 31.75 |
| Lang | Ochiai | 24 | 44 | 50 | 4.63 | – |
| | FLUCCS | 40 | 53 | 55 | 3.4 | 18.2 |
| | DeepFL | 46 | 54 | 59 | 2.15 | 3.36 |
| | GRACE | 46 | 54 | 55 | 2.34 | 85.2 |
| | GNet4FL | 42 | 55 | 57 | 2.01 | 102.86 |
| Math | Ochiai | 23 | 52 | 62 | 9.73 | – |
| | FLUCCS | 48 | 77 | 83 | 4.64 | 17.28 |
| | DeepFL | 63 | 85 | 91 | 3.76 | 8.68 |
| | GRACE | 60 | 81 | 91 | 3.46 | 352.11 |
| | GNet4FL | 67 | 82 | 88 | 3.32 | 238.04 |
| Overall | Ochiai | 66 | 135 | 158 | 11.91 | – |
| | FLUCCS | 118 | 183 | 198 | 7.98 | 13.97 |
| | DeepFL | 146 | 195 | 209 | 7.51 | 7.90 |
| | GRACE | 148 | 193 | 214 | 5.3 | – |
| | GNet4FL | 160 | 204 | 217 | 5.38 | 95.44 |

statistically significant difference between the two compared methods. The $p$ values of GNet4FL with Ochiai and FLUCCS are less than 0.05, at 0.019 and 0.037 respectively. This indicates that GNet4FL is an effective approach for fault localization.

By further analyzing the results obtained by GNet4FL, we consider that satisfactory experimental results are mainly due to the training and testing of the same project. During the experiments, the training set and the test set are from the same project, which allows the model to better predict the faults in this project. For Example, we have a MAR of 9.52, which is 8% higher than Grace in `Time`. In this project, more fault types are logical faults, leading to a lower ranking for individual faults caused by omissions. The graph in Grace contains test nodes and source code nodes by constructing similarities between them. The node attributes consist of node type

and similarity, while GNet4FL uses the test results directly as part of the node attributes. GNet4FL counts the times covered by the failed test cases. In previous studies of the SBFL approach (Jones and Harrold 2005; Ju et al. 2014), they proposed that the execution results of test cases are associated with suspiciousness. If a program entity executes more passed test cases, the lower suspiciousness is. Conversely, the more failed test cases it executes, the higher suspiciousness it should be. If a failed test case does not cover a node, its final probability will be close to 0 after training. We believe that the above conclusion is also beneficial for AST node classification.

Besides, the results show that GNet4FL performs better on the Mockito project. Further analysis of each version in this project showed that 47.3% of the faults occurred in the loop or block statements. These statements are exactly the focus during the preprocessing of ASTs in our research, and there are 494 corresponding statements in the Mockito project. Conversely, these types account for 38.7% of the Math project, and most faults occur in declaration or assignment statements. For example, the fault report for Math-35 shows a fault in the assignment statement on line 51. Thus, we think the preprocessing of critical statements is one of the reasons why GNet4FL is better than Grace.

We also collect the execution time of each method and calculate the training time (ignoring data preprocessing and testing time) for each version, with DeepFL, Grace, and GNet4FL set to 50 epochs, and FLUCCS set to 50 generations. In our empirical study, we found that the model of each method reaches optimum performance in the epoch range of 30–60. Therefore, we set the number of epochs to 50. The execution time of all methods does not needed for the following two reasons: (1) the execution time of Ochiai is mostly the time of the test cases, and we only count the training time; (2) the graph structure of Grace is large, and in some projects, it can only be run with small batch size. And when the batch size is set small, it will affect the convergence of the model. Therefore, during the training process, we only experimented on two projects (Math, Lang), where the batch size was set to 20 and 30, respectively.

The last column of Table 3 shows the model training time, where '-' indicates that its training time is not recorded. For each project, the average training time per faulty version is used as the result in this paper. The results show that the training time using graph neural networks is much longer than that of MLP and SVM. Moreover, we observed that Grace required over 30 G of memory in the full-batch and 10 G of memory in the batch size of 10, compared to GNet4FL which required only 16 G of memory. Therefore, GNet4FL has a lower computational cost, but the result is close to Grace.

To better compare the efficiency of GNet4FL and Grace, this paper compares their running times under two GPU memories. The parameters such as batch size, epoch, and learning rate are kept consistent during the experiments. Table 4 shows the time costs of model training for each subject. Column 2 is the result with 12 G GPU memory and column 3 is the result with 16 G GPU memory. From the table, GNet4FL is lower than Grace method for all projects except one. Specifically, GNet4FL is only 43.65% of Grace in the Time project. This means that GNet4FL is faster than Grace by more than half. We analyze two reasons for this performance difference. Firstly, the running efficiency of GNN is usually related to the number of nodes and edges in the graph. This paper constructs a smaller-scale graph structure that benefits the model computation. It uses the results of failed test cases directly as one of the node features,

**Table 4** Efficiency of different techniques

| Subject | Technique | 12 G(s) | 16 G(s) |
|---|---|---|---|
| Chart | Grace | 265.5 | 167.33 |
| | GNet4FL | 145.65 | 77.19 |
| Time | Grace | 280.5 | 167.97 |
| | GNet4FL | 122.44 | 73.88 |
| Mockito | Grace | 262.12 | 153.97 |
| | GNet4FL | 120.14 | 62.02 |
| Lang | Grace | 200.14 | 136.6 |
| | GNet4FL | 271.63 | 175.42 |
| Math | Grace | 384.6 | 196.58 |
| | GNet4FL | 262.79 | 160.32 |

combining static and dynamic features of the code while reducing the size of the graph structure. The Grace method incorporates information of test cases as part of the nodes in the graph, calculates the similarity between test case nodes and AST node names, and uses it as one of the node features. This increases the number of nodes and edges. Secondly, GraphSAGE computation speed is faster than that of Gated Graph Neural Networks (GGNN), contributing to the faster execution time of GNet4FL. To verify this conjecture, this paper changes the GraphSAGE method to GGNN and computes its running time under 12 G GPU memory. The results show that the time of GGNN in the Time project is 170.23, which is 39.03% more than GraphSAGE.

**Summary for RQ1:** Compared with the baselines, GNet4FL can locate more faults than the baselines in terms of Top-$k$ ($k = 1, 3, 5$) and achieve a lower score in MAR.

### 5.2 Result analysis for RQ2

To verify the effect of GNN on our framework, we invested GNet4FL with traditional machine learning techniques, i.e., removing GraphSAGE in step 2 and replacing the method in step 3 with the model to compare. To answer RQ2, we selected three baselines, including K-Nearest Neighbor (KNN), multi-layer perceptron (MLP), and Random Forest (RF). We implemented these algorithms with scikit-learn library and tuned the hyper-parameters with NNI to obtain optimal comparison.

Figure 3 illustrated the number of located faults by four approaches (i.e., GNet4FL, KNN, MLP, RF) within Top-1, Top-3, and Top-5 entities. As shown in Fig. 3, GNet4FL significantly outperformed the other three baselines by 58. 39%, 72.22% and 9.6% respectively. Figure 4 presents a box plot of MAR obtained by four techniques across the five projects. Comparing the four techniques in the Figure, GNet4FL has the lowest average cost, followed by RF. Among the three machine learning methods, RF performs relatively well. It locates 72 more faults compared to KNN. Specifically, RF is only 1 less than GNet4FL in Top-1 for small projects (e.g. Mockito) and more than 5 less than GNet4FL in Top-1

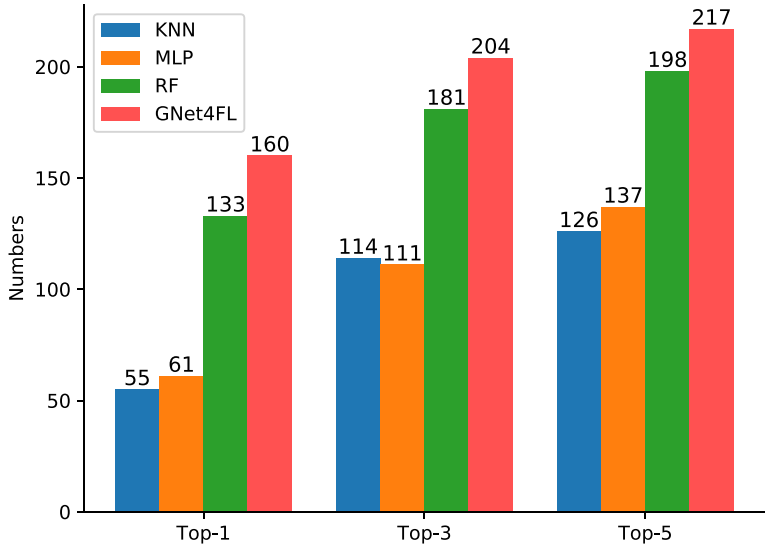


**Fig. 3** Effectiveness of 4 approaches

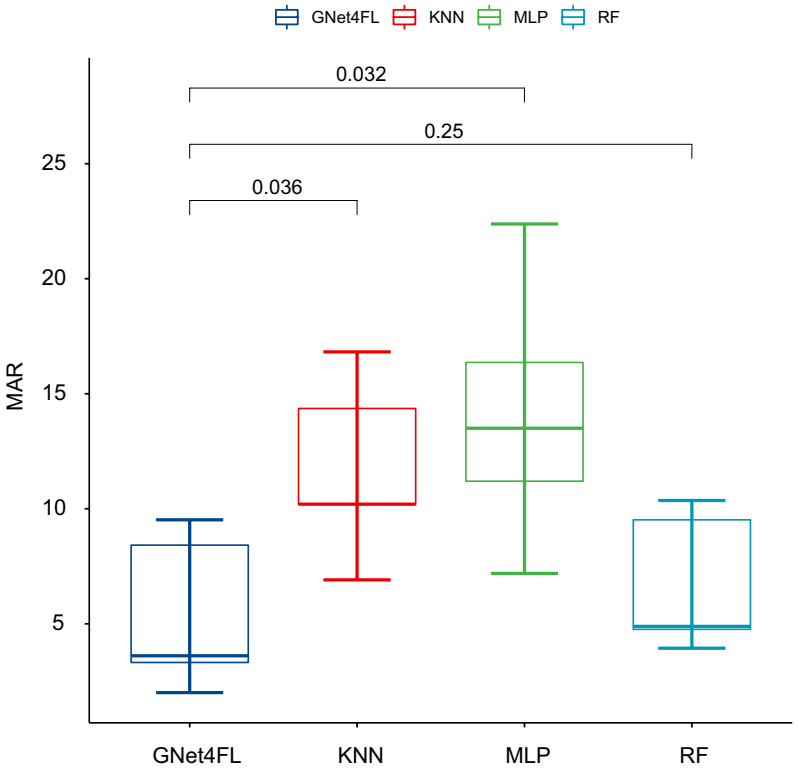


**Fig. 4** MAR and *p* value of compared approaches

for large projects (e.g. Chart, Math). GNet4FL outperforms RF due to its ability to consider contextual information of the code, whereas RF is limited to code sequence information. Moreover, RF is better suited for smaller datasets. When there is too much data, RF takes more time to compute the feature values and is less efficient. GraphSAGE can be trained on subgraphs by sampling, which requires less memory and is more efficient. It can also better handle complex relationships between AST nodes.

Furthermore, we calculated the $p$ value to detect the differences between GNet4FL and three baselines (i.e., KNN, MLP, RF) by Wilcoxon rank-sum test and corrected the obtained $p$ values by using Benjamini_Hochberg (BH) method (Yang et al. 2016). We set the significance level at 0.05, which means that if the corrected $p$ values are less than 0.05. The result indicates a significant difference between the two compared methods.

By examining $p$ values shown in Fig. 4, we can find that two $p$ values are less than 0.05. The result indicates significant differences between GNet4FL and two compared baselines (i.e., KNN and MLP). That is to say, there are significant differences between graph neural network-based and traditional machine learning-based approaches (i.e., KNN and MLP). But the $p$ value between GNet4FL and RF is 0.25, which means there is no significant difference between them. Therefore, we can conclude that graph neural networks (GNN) help better fault localization because Graph representation learning can effectively extract semantic information and features.

**Summary for RQ2:** Graph neural networks can extract graph structure information well and improve the fault locating performance of GNet4FL.

### 5.3 Result analysis for RQ3

GNet4FL introduces coverage information as a node attribute and prunes the AST nodes to reduce redundant information based on static analysis. To verify whether these two processes positively affect the model, we analyzed the effects of both methods before and after their use.

To simplify our discussion, we give three notations to represent three scenarios, namely GNet4FL, GNet4FL-prune, and GNet4FL-SBFL, whose detailed description are give as follows:

- GNet4FL means using our framework (GNet4FL) with both prune and SBFL.
- GNet4FL-prune means using our framework (GNet4FL) without pruning the AST.
- GNet4FL-SBFL means that using our framework (GNet4FL) without coverage information, and only word vectors are used as node attributes.

Similar to the process of answering RQ2, we used Top-$k$ and MAR to evaluate the model performance and $p$ value to find significant differences between the compared techniques.

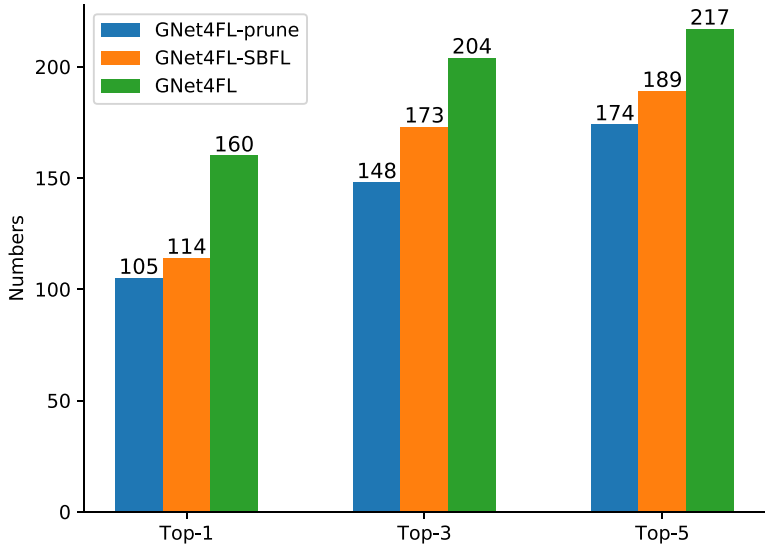


**Fig. 5** Effectiveness of 3 approaches

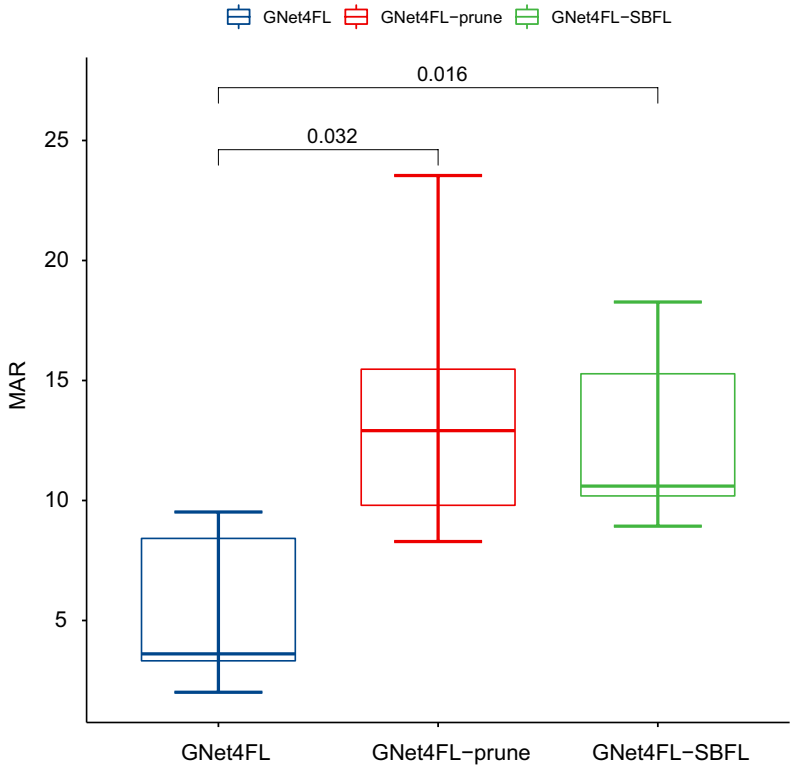


**Fig. 6** MAR and *p* value of compared approaches

**Table 5** Cross-project effectiveness on Defects4J

| Subject | Top-1 | Top-3 | Top-5 |
|---|---|---|---|
| Chart | 5 | 8 | 12 |
| Time | 7 | 9 | 13 |
| Mockito | 12 | 14 | 19 |
| Lang | 21 | 25 | 30 |
| Math | 28 | 34 | 40 |
| Overall | 73 | 90 | 114 |

Figure 5 illustrated the number of faults located by three approaches (i.e., GNet4FL, GNet4FL-prune, and GNet4FL-SBFL) within Top-1, Top-3, and Top-5 entities. For example, GNet4FL can locate 217 faults on Top-5, outperforming GNet4FL-prune and GNet4FL-SBFL by 14.81% and 24.71%. Furthermore, we calculated $p$ value to detect the differences between GNet4FL,GNet4FL-prune, and GNet4FL-SBFL) by Wilcoxon rank-sum test, and corrected the obtained $p$ values by using Benjamini_ Hochberg (BH) method (Yang et al. 2016). We set the significance level at 0.05, which means that if the corrected $p$ values are less than 0.05. The result indicates a significant difference between the two compared techniques. Comparing GNet4FL-prune and GNet4FL-SBFL with GNet4FL, it can be seen from Fig. 6 that the $p$ values between them are 0.032 and 0.016, respectively. At a significance level of 0.05, their $p$ values are both below 0.05. This means that the preprocessing of the AST and taking coverage information as one of the attributes has significantly improved.

**Summary for RQ3:** Each part of the method plays an important role and can improve performance by up to 24%.

### 5.4 Result analysis for RQ4

We obtained data from the same project to train and answer the previous three research questions. However, the same project often belongs to the same type of program. For example, Apache commons-Math is a computational library that provides data computation components; JfreeChart provides java drawing components. This can lead to problems where it performs well in the same project and poorly in other projects. To overcome this problem, we investigated the effectiveness of GNet4FL in cross-project. We obtained four project data to train and invest in the last research question (RQ4).

The efficiency of GNet4FL on cross-project is shown in Table 5. Compared to the results in RQ1, the results for cross-project are significantly worse than within-project. GNet4FL is 45.62% of the results for within-project on Top-1. Overall, the results in Top-1, Top-3, and Top-5 are 73, 90, and 114, respectively. Specifically, the results in Chart, Time, Mockito, Lang, and Math are 12, 13, 19, 30, and 40 in Top-5, which are 52%, 65%, 66%, 53%, 45% of the results in within-project. Compared with it, DeepFL locates 9, 15, and 17 within Top-1, Top-3, and Top-5 in Mockito

projects. GNet4FL is 33.3% higher on Top-1. But GNet4FL is inferior to DeepFL in other projects. Especially on Lang, DeepF locates 37, 45, and 49, which is 16 more than GNet4FL in Top-1. Combining the code characteristics of each project, we found that GNet4FL works better in projects with fewer lines of code.

**Summary for RQ4:** In some cases, GNet4FL can still have encouraging results in cross-project predictions, reaching 65% of within-project predictions on Top-5.

## 6 Threats to validity

This section will discuss potential threats to our research's validity, including internal and external validity.

*Threats to internal validity:* The internal threat mainly lies in the implementation of the proposed approach. Each component of our framework, such as AST pruning, GraphSAGE, etc., can obtain incorrect data due to calculation errors. To mitigate this threat, We use established and widely used mature frameworks or libraries, such as PyTorch, Javalang, and Gensim. Moreover, we apply code open-source as the baseline during comparison. Furthermore, we employ NNI, a widely used hyper-parameters tuning tool in numerous Neural networks research, to obtain our model's optimal hyper-parameters.

*Threats to external validity*: The most significant external threat to our experimental subject lies in the dataset Defects4J. To mitigate this threat, we selected and validated several state-of-the-art methods using Top-$k$ and MAR as evaluation metrics. Due to the randomness of the graph neural network in the division of the data set, which caused inconsistent results, we ran it ten times and took the mean value as the final result.

## 7 Conclusion

This paper presents GNet4FL, a method based on a graph convolutional neural network that learns latent information from graph structures. The performance of GNet4FL outperforms all current techniques. In particular, GNet4FL succeeded in locating 160 faults in Top-1, while the best method only located 148. We also analyzed the three components in the technique and found that all had a positive effect.

Our current work is encouraging, yet still limited. In the future, it can be developed in the following ways: (1) Finding a more suitable pruning method for ASTs, such as combining program slicing to reduce the number of program entities prior to AST creation. (2) Doing further research into how graph neural networks can be paired with fault localization. (3) Examining if there is any correlation between the faults in different projects.

**Author contributions** JQ and XJ wrote the main manuscript text and figures, and XC helped to improve the approach and the framework. All authors reviewed the manuscript.

## Declarations

**Conflict of interest** The authors declare no competing interests.

## References


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

Abreu, R., Zoeteweij, P., Golsteijn, R., Van Gemund, A.J.: A practical evaluation of spectrum-based fault localization. J. Syst. Softw. **82**(11), 1780–1792 (2009)

Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: learning distributed representations of code. In: Proceedings of the ACM on Programming Languages 3(POPL), pp. 1–29 (2019)

Anowar, F., Sadaoui, S., Selim, B.: Conceptual and empirical comparison of dimensionality reduction algorithms (pca, kpca, lda, mds, svd, lle, isomap, le, ica, t-sne). Comput. Sci. Rev. **40**, 100378 (2021)

Bruna, J., Zaremba, W., Szlam, A., LeCun, Y.: Spectral networks and locally connected networks on graphs (2013). arXiv preprint arXiv:1312.6203

Chung, H.M., Gey, F., Piramuthu, S.: Data mining and information retrieval. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, vol. 7, pp. 841–842. IEEE Computer Society (2002)

Dutta, A., Godboley, S.: Msfl: a model for fault localization using mutation-spectra technique. In: International Conference on Lean and Agile Software Development, pp. 156–173 (2021). Springer

Feyzi, F.: CGT-FL: using cooperative game theory to effective fault localization in presence of coincidental correctness. Empir. Softw. Eng. **25**(5), 3873–3927 (2020)

Fushiki, T.: Estimation of prediction error by using k-fold cross-validation. Stat. Comput. **21**(2), 137–146 (2011)

Gong, P., Zhao, R., Li, Z.: 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), pp. 1–10. IEEE (2015)

Gu, W., Tandon, A., Ahn, Y.-Y., Radicchi, F.: Principled approach to the selection of the embedding dimension of networks. Nat. Commun. **12**(1), 1–10 (2021)

Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, pp. 1025–1035 (2017)

Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 273–282 (2005)

Ju, X., Jiang, S., Chen, X., Wang, X., Zhang, Y., Cao, H.: HSFal: effective fault localization using hybrid spectrum of full slices and execution slices. J. Syst. Softw. **90**, 3–17 (2014)

Keller, F., Grunske, L., Heiden, S., Filieri, A., van Hoorn, A., Lo, D.: A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 114–125. IEEE (2017)

Kipf, T.N., Welling, M.: Semi-supervised Learning with Graph Convolutional Networks. ICLR (2017)

Küçük, Y., Henderson, T.A., Podgurski, A.: Improving fault localization by integrating value and predicate based causal inference techniques. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 649–660. IEEE (2021)

Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 218–229 (2017). IEEE

Li, X., Zhang, L.: Transforming programs and tests in tandem for fault localization. In: Proceedings of the ACM on Programming Languages 1(OOPSLA), pp. 1–30 (2017)

Li, X., Li, W., Zhang, Y., Zhang, L.: DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 169–180 (2019)

Li, Y., Wang, S., Nguyen, T.N.: Fault localization with code coverage representation learning. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 661–673. IEEE (2021)

Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: SOBER: statistical model-based bug localization. SIGSOFT Softw. Eng. Notes **30**(5), 286–295 (2005). https://doi.org/10.1145/1095430.1081753

Liu, Y., Li, Z., Wang, L., Hu, Z., Zhao, R.: Statement-oriented mutant reduction strategy for mutation based fault localization. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 126–137. IEEE (2017)

Lou, Y., Zhu, Q., Dong, J., Li, X., Sun, Z., Hao, D., Zhang, L., Zhang, L.: Boosting coverage-based fault localization via graph-based representation learning. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 664–676 (2021)

Mao, X., Lei, Y., Dai, Z., Qi, Y., Wang, C.: Slice-based statistical fault localization. J. Syst. Softw. **89**, 51–62 (2014)

Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. (TOSEM) **20**(3), 1–32 (2011)

Papadakis, M., Le Traon, Y.: Metallaxis-FL: mutation-based fault localization. Softw. Test. Verif. Reliab. **25**(5–7), 605–628 (2015)

Park, S., Vuduc, R.W., Harrold, M.J.: Falcon: fault localization in concurrent programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 245–254 (2010)

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 609–620 (2017). IEEE

Peng, Z., Xiao, X., Hu, G., Sangaiah, A.K., Atiquzzaman, M., Xia, S.: Abfl: an autoencoder based practical approach for software fault localization. Inf. Sci. **510**, 108–121 (2020)

Planning, S.: The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology (2002)

Qian, J., Ju, X., Chen, X., Shen, H., Shen, Y.: AGFL: a graph convolutional neural network-based method for fault localization. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 672–680 (2021). IEEE

Raunak, V.: Simple and effective dimensionality reduction for word embeddings (2017). arXiv preprint arXiv:1708.03629

Sohn, J., Yoo, S.: Fluccs: Using code and change metrics to improve fault localization. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 273–283 (2017)

Soremekun, E., Kirschner, L., Böhme, M., Zeller, A.: Locating faults with program slicing: an empirical analysis. Empir. Softw. Eng. **26**(3), 1–45 (2021)

Van der Maaten, L., Hinton, G.: Visualizing data using t-SNE. J. Mach. Learn. Res. **9**(11), 2579–2605 (2008)

Vancsics, B., Horváth, F., Szatmári, A., Beszédes, Á.: Fault localization using function call frequencies. J. Syst. Softw. **193**, 111429 (2022)

Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., Yu, P.S.: Heterogeneous graph attention network. In: The World Wide Web Conference, pp. 2022–2032 (2019)

Wong, W.E., Qi, Y.: BP neural network-based effective fault localization. Int. J. Softw. Eng. Knowl. Eng. **19**(04), 573–597 (2009)

Wong, W.E., Debroy, V., Choi, B.: A family of code coverage-based heuristics for effective fault localization. J. Syst. Softw. **83**(2), 188–208 (2010). https://doi.org/10.1016/j.jss.2009.09.037

Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. IEEE Trans. Reliab. **63**(1), 290–308 (2013)

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **42**(8), 707–740 (2016)

Woolson, R.F.: Wilcoxon signed-rank test. In: Wiley Encyclopedia of Clinical Trials, pp. 1–3 (2007)

Wu, F., Souza, A., Zhang, T., Fifty, C., Yu, T., Weinberger, K.: Simplifying graph convolutional networks. In: International Conference on Machine Learning, pp. 6861–6871. PMLR (2019)

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. **32**(1), 4–24 (2020)

Xiao, X., Pan, Y., Zhang, B., Hu, G., Li, Q., Lu, R.: ALBFL: a novel neural ranking model for software fault localization via combining static and dynamic features. Inf. Softw. Technol. **139**, 106653 (2021)
Xie, X., Chen, T.Y., Kuo, F.-C., Xu, B.: A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. (TOSEM) **22**(4), 1–40 (2013a)
Xie, X., Wong, W.E., Chen, T.Y., Xu, B.: Metamorphic slice: an application in spectrum-based fault localization. Inf. Softw. Technol. **55**(5), 866–879 (2013b)
Xu, J., Wang, F., Ai, J.: Defect prediction with semantics and context features of codes based on graph representation learning. IEEE Trans. Reliab. **70**(2), 613–625 (2020)
Xuan, J., Jiang, H., Hu, Y., Ren, Z., Zou, W., Luo, Z., Wu, X.: Towards effective bug triage with software data reduction techniques. IEEE Trans. Knowl. Data Eng. **27**(1), 264–280 (2014)
Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H.: Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 157–168 (2016)
Yang, Y., Xia, X., Lo, D., Grundy, J.: A survey on deep learning for software engineering. ACM Comput. Surv. (CSUR) (2021). https://doi.org/10.1145/3505243
Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 783–794. IEEE (2019a)
Zhang, M., Li, Y., Li, X., Chen, L., Zhang, Y., Zhang, L., Khurshid, S.: An empirical study of boosting spectrum-based fault localization via pagerank. IEEE Trans. Softw. Eng. **47**(6), 1089–1113 (2019b)
Zhang, J.M., Harman, M., Ma, L., Liu, Y.: Machine learning testing: survey, landscapes and horizons. IEEE Trans. Softw. Eng. **48**, 1–36 (2020)
Zhang, Z., Lei, Y., Mao, X., Yan, M., Xu, L., Zhang, X.: A study of effectiveness of deep learning in locating real faults. Inf. Softw. Technol. **131**, 106486 (2021)
Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 14–24 (2012). IEEE
Zou, D., Liang, J., Xiong, Y., Ernst, M.D., Zhang, L.: An empirical study of fault localization families and their combinations. IEEE Trans. Softw. Eng. **47**(2), 332–347 (2019)



## Authors and Affiliations

**Jie Qian[1] · Xiaolin Ju[1] · Xiang Chen[1]**

Jie Qian
qianjieee@gmail.com

Xiang Chen
xchencs@ntu.edu.cn

[1] School of Information Science and Technology, Nantong University, Nantong 226019, Jiangsu, China