Automatic bug localization using a combination of deep learning and model transformation through node classification

Leila Yousofvand[1], Seyfollah Soleimani[1,*], Vahid Rafe[1,2]

l.yousofvand@gmail.com
s-soleimani@araku.ac.ir
v-rafe@araku.ac.ir, v.rafe@gold.ac.uk

Abstract

Bug localization is the task of automatically locating suspicious commands in the source code. Many automated bug localization approaches have been proposed for reducing costs and speeding up the bug localization process. These approaches allow developers to focus on critical commands. In this paper, we propose to treat the bug localization problem as a node classification problem. As in the existing training sets, the whole graphs are labeled as buggy and bug-free, it is required first to label all nodes in each graph. To do this, we use the Gumtree algorithm, which label the nodes by comparing the buggy graphs with their corresponding fixed graphs. In classification, we propose to use a type of Graph neural networks (GNNs), GraphSAGE. The used dataset for training and testing is JavaScript buggy codes and their corresponding fixed codes. The results demonstrate that the proposed method outperforms other related methods.

Keywords: Deep learning, Bug localization, Node classification, Graph neural networks.

1. Introduction

Software developers have helped simplify life, save time and money, and help connect with others by developing and producing software. When software bugs occur during software development and maintenance, accurate and timely bug localization is an important guarantee of software quality. Today, it is impossible to quickly locate bugs in case of failure because the complexity and scale of modern software have increased. Bug localization has been a manual task that is time-consuming and very expensive [1]. Manual bug localization is dependent on the experience, intuition, and judgment of the software developer to prioritize and detect probable buggy code. Thus, the demand for techniques that are able to identify bug locations in a program with minimal human intervention has increased. Several tools such as FindBugs [2], PMD [3], TAJS [4], and ESLint [5] have been widely used in software debugging as auxiliary tools for detecting and locating bugs in Java and JavaScript languages. Despite the success of these techniques, developers need more advanced features and capabilities. For instance, if bug reports contain misleading explanations, information retrieval (IR) approaches are less effective [6]. Spectrum-based approaches are less effective in locating multiple bugs [7]. With this in mind, we present a graph-based approach that can locate multiple bugs. In this approach, we have designed a node classifier that uses real buggy code and corresponding graph as input and classifies graph nodes into buggy and bug-free.

Graphs are data structures used to solve many problems in mathematics and computer science. The most important use of graphs is to model various phenomena and relationships in the real world. For example, Google Maps uses graphs to model the road network and route to the final destination. Facebook is a graph that shows people and friendships between them. The Internet is a huge graph that shows pages

[1] Department of Computer Engineering, Faculty of Engineering, Arak University, Arak 38156-8-8349, Iran
[2] Department of Computing, Goldsmiths University of London, London, UK
* Corresponding author, s-soleimani@araku.ac.ir

and links between pages. All objects in physical space, human relations, document structures, semantic structures, geographical or physical models can be represented using graphs. In recent years, graphs have been used to model program source code [8, 9]. There is valuable information about the syntactic and semantic structure of the program in this type of graph. Since a large number of software bugs are related to the syntactic and semantic structure of program commands, we map the input program source code to a graph representation and then train a node classifier to find bugs. To make a classifier, a Graph neural network (GNN) [10], is used. GNN is a beneficial deep learning method that provides a simple way to perform prediction tasks at the graph, node, and edge levels [11]. It summarizes and gather information from graph structures and capture graph dependencies by sending messages between sets of objects (nodes and edges) [12]. There are several types of GNNs can be used depending on the type of graph or data required. Graph Attention Network (GAN) [13], Convolution Graph Network (GCN) [14], Graph SAmple and aggreGatE (GraphSAGE) [15] are variants of GNNs. GCN [14] has been used in various issues due to its high efficiency and simplicity and has attracted more attention. They have been very successful in graphic subjects such as graph classification [16] and node classification [14], and so on. For this reason, we decide to use GCN model for our classification step. But a major drawback of GCN is scalability. In general, each node's feature vector depends on its entire neighborhood. This can be quite inefficient for huge graphs with big neighborhoods. To solve this issue, we use GraphSAGE, which is an improvement of GCN. It handles this problem by sampling a subset of neighborhoods to conduct propagation instead of using all neighborhoods information.

There are two challenges to graph-based bug localization: small training set and imbalanced dataset. When a training dataset is small, it is impossible to learn a reliable classifier because performing too many repetitions can lead to overfitting. To handle this challenge, we use thousands of real buggy codes and related bug-free codes. Researchers have shown that most lines in buggy source code are bug-free, and bugs make up only a small portion of a program's source code [17]. So, the training set is imbalanced. To overcome this challenge, we use a re-sampling technique to generate additional nodes from the minor class, buggy nodes, and then train the model.

We propose to use GraphSAGE model and node classification for automatic bug localization. To the best of our knowledge, this is the first article that uses this graph representation and GraphSAGE model for this problem. We will employ JavaScript code which has been the most popular language on GitHub for the past eight years [18]. In our proposed method, we transform each input code into a graph based on Abstract Syntax Tree (AST). The main contributions of our method are preparing labeled training data, using deep learning to train a node classifier, and handling imbalanced classification in the graph data structure for bug localization in JavaScript codes. We use Gumtree algorithm [19] to label nodes of graphs by comparing the nodes of the buggy graphs with corresponding fixed graphs. In addition, we tested several configurations of the node classification to select the best one. Our method covers a broader range of bugs such as undefined attributes, adding the async keyword, incorrect use of const /let /var quantifiers, adding the export keyword, varnaming, varmisuse, etc. Also, it works well for more complex bugs that need more than one fix.

The rest of the paper is organized as follows: In Section 2, a survey of related work is presented. In Section 3, the required background, such as bugs in JavaScript program, Abstract Syntax Tree, Graph Convolutional Network, GraphSAGE, and imbalanced data classification are described. Our proposed method is detailed in Section 4. The experimental setup, including dataset, evaluation metrics, and experimental settings are presented in Section 5. Section 6 discuss the experimental results. Finally, we conclude with future work in Section 7.

2. Related Works

2.1. Spectrum-based bug localization

The spectrum-based bug localization techniques [20] rank elements of the program according to their probability of being buggy based on failing and passing tests and the analysis of the program elements [21]. Elements that have been performed by many failing tests and a small number of passing tests are likely to have bugs; conversely, elements of the program that have been performed by many passing tests and a small number of failing tests are likely to be bug-free. In early spectrum-based bug localization studies [22, 23], only failed test cases were used. Subsequent studies achieve better results using passing and failing experiments [24]. Suspicious values of code lines can be calculated using various criteria [25]. Spectrum-based bug localization techniques use four criteria, ,Tarantula [26], GenProg [27], Ochiai [28], and Jaccard [29], to calculate the suspiciousness of code lines in bug localization [30]. Researchers have investigated more advanced cases, such as using neural networks to predict buggy code [31] or reducing the number of test cases [32]. Spectrum-based approaches depend on experimental inputs for their performance, but our approach does not require experimental inputs and is based on the analysis of entities and their relationships, which can complement spectrum-based approaches.

2.2. IR-based bug localization

In IR-based bug localization techniques buggy files are identified by using the textual similarity between bug reports and the code. These methods consider a report of the bug as a query and turn bug localization into a search problem. Some machine learning techniques such as Latent Dirichlet Allocation [33], Support vector machine [34], Naive Bayes [35], and Latent Semantic Analysis [36], are used as a similarity to find the snippet code that is related to a special bug report. Several code structures such as methods, class, comments, and variables have been exploited from source code for bug localization in [37]. In addition to bug reports, version histories [38] have also been studied. Although we extracted features from equivalent bug and non-bug files for labeling, our approach is not IR-based and we do not calculate the similarity between bug reports and source files in our approach. Our approach can also be used as a complement to IR-based approaches.

2.3. Machine learning-based bug localization

The goal of Machine learning-based bug localization techniques is inferring or learning the bug location. Researchers proposed an approach based on a back-propagation neural network [39]. In this neural network, the difference between the output obtained from the network with the desired output in the training data set is calculated as suspicious value. Execution results and coverage data for each test case are gathered and used to train a neural network. A generalized back-propagation approach to the object-oriented context is proposed [40]. This study also investigated the use of support vector machines (SVMs). Another approach is proposed based on radial basis function(RBF) networks [41]. The RBF networks are less sensitive to local minima problem and have a better learning rate [42]. When the training phase of the RBF network is completed, the output is considered as suspicious of the statement. These approaches differ from our approach by using different techniques as well as taking different inputs from our approach.

2.4. Model-based bug localization

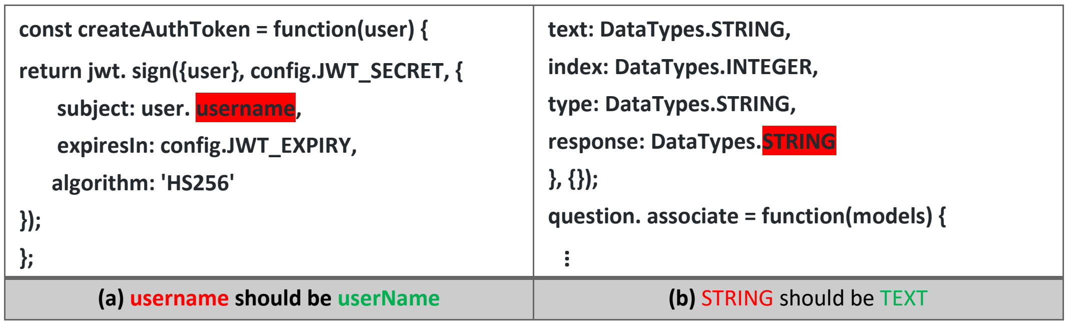
In Model-based bug localization techniques, a model can define bug signatures. These techniques use the differences between the observed behaviors of the source code and the behaviors of the model to locate bugs in the source code [43]. Some bug localization approaches use models constructed directly from the actual source codes [44, 45]. Actual source code means that these source codes may contain bugs. Dependency-based model derives from dependencies between program statement. A dependency model which can handle features of the Java language, such as methods, classes, while-loops, conditionals, and assignments is proposed in [46]. In another research, dependency-based models are constructed based on first-order logic and the differences between test cases and models are used to identify suspicious expressions [47]. In [48], researchers have built program dependency graphs from the bug fixes and extracted the feature vector of each node by graph analysis. They combined graph analysis for finding the bug location. Value-based models are applied to locate buggy components [49]. This type of model represents data-flow information in programs. Due to the higher computational complexity compared to dependency-based models, they use in small projects [50]. A graph-based code representation that modeled the syntactic and semantic structure of program commands is proposed in [8]. They used a pointer network [51] into the graph structure. These approaches differ from our approach by using different types of graphs. Also, most of the methods in this category cover specific types of bugs or target bugs in specific databases. Our approach covers a wider range of bugs such as undefined attributes, adding the async keyword, incorrect use of const /let /var quantifiers, adding the export keyword, varname, varmisuse, etc. It also works well for more complex bugs that need to be fixed in multiple places.

3. Preliminaries

3.1. Bug in JavaScript program

A software bug is a defect or an error in a source code program that produces unexpected results. Software bugs can lead to delays in software projects and increase software maintenance costs.

The JavaScript programming language is widely used in client-side web applications. This programming language has been the most used language in GitHub for the past eight years [18]. Although JavaScript is very popular with developers, inherent features such as dynamic typing and runtime evaluation make it a vulnerable programming languages. Fig. 1 shows examples of bugs in JavaScript code.



```
const createAuthToken = function(user) {
return jwt. sign({user}, config.JWT_SECRET, {
     subject: user. username,
     expiresIn: config.JWT_EXPIRY,
    algorithm: 'HS256'
});
};
```

(a) username should be userName

```
text: DataTypes.STRING,
index: DataTypes.INTEGER,
type: DataTypes.STRING,
response: DataTypes.STRING
}, {});
question. associate = function(models) {
  ⋮
```

(b) STRING should be TEXT

Fig. 1. Two examples of JavaScript buggy code snippets

3.2. Abstract Syntax Tree

The Abstract Syntax Tree (AST) is a tree representation of the logical structure of a statement. It is similar to the parsing tree. The parse tree contains all the grammatical symbols (terminals and non-terminals) encountered during parsing. The abstract syntax tree is “abstract”, meaning that it does not show all the details that appear in the actual syntax, but only contains the content-related and structural details [52]. The following tasks are needed to extract an abstract syntax tree from a parse tree: (i) Remove Separators and priority markers such as parentheses. (ii) Replace parents that have only one child with their child. (iii) Replace the remaining non-terminals by operators that are their children.

3.3. Graph Convolutional Network

Graph Convolutional networks (GCNs) are an attempt to apply deep learning techniques to graphs. The most important part of GCN is a graph. A graph is a data structure consisting of two components: nodes and the relationships between nodes, known as edges. With N nodes, it can be represented as G = (V, E), where V = $\{v_i \mid i = 1\ldots N\}$ is a set of nodes and E ⊆ V × V is a set of edges. A ∈ $R^{N\times N}$ denotes the adjacency matrix of the graph, and the node feature matrix in this graph is represented by X ∈ $R^{N\times D}$. In our notation, we use M(i) to represent the set of neighboring nodes of node i.

GCN is as an beneficial semi-supervised learning on graphs that uses a combination of node features and information of graph structure [14]. A graph convolution layer is defined as follows:

$$H_i = f\left(\widetilde{D}^{-1}\tilde{A}X_iW\right) \qquad (1)$$

The graph convolution consisting of four phases: (i) applying a linear feature transformation to X by multiplying it by W (XW) and sharing W (a matrix of trainable parameters of graph) weights between all nodes. (ii) Propagating the information of node to adjacent vertices as well as the node itself by applying $\tilde{A}$Y, where Y = XW and $(\tilde{A}Y)_i = \sum_j \tilde{A}_{ij} Y_j = Y_i + \sum_{j\in M(i)} Y_j$. (iii) Normalizing each row i and keeping a fixed feature scale after graph convolution by multiplying $\tilde{A}$Y by $\widetilde{D}^{-1}$. (iv) Applying a nonlinear activation function *f* to output the graph convolution results. The GCN propagation rule is defined as follows:

$$H_i^{(l+1)} = \sigma\left(LH_i^{(l)}W^{(l)}\right) \qquad (2)$$

where, $H_i^{(l)}$ is the node embedding representation matrix for a node i at the l-th layer. We initialize $H_i^{(0)} = X_i$. σ (·) is the activation function such as ReLU (·) = max (0, ·). $L \in R^{N\times \mathrm{N}}$ is the aggregation matrix. GCN [14] uses the graph Laplacian L=Â≜ $\widetilde{D}^{-1/2}\hat{A}\widetilde{D}^{-1/2}$ as the aggregator where $\widetilde{D} = diag(\sum_i \tilde{A}_{1i}, \ldots, \sum_i \tilde{A}_{Ni})$, $\hat{A} = A + I_N$. I_N is the identity matrix.

3.4. GraphSAGE (SAmple and aggreGatE)

One of the significant drawbacks of GCN architecture is scalability. That is, the characteristic vector of each node depends on its entire neighborhood. This can be quite inefficient for big graphs with big neighborhoods. To solve this problem, GraphSAGE has been proposed. The main idea of GraphSAGE is to sample the neighborhood information set instead of using the whole of them for propagation. The GraphSAGE algorithm [15] is a comprehensive upgrade over the original GCN. The GraphSAGE forward propagation rule is defined as follows:

$$H_{M(i)}{}^{(l+1)} = aggregate\left(H_j{}^{(l)}, \forall\, \mathrm{j} \in \mathrm{M(i)}\right)$$
$$H_i{}^{(l+1)} = \sigma\left(W \cdot concat(H_i{}^{(l)}, H_{M(i)}{}^{(l+1)})\right) \qquad (3)$$
$$H_i{}^{(l+1)} = norm(H_i{}^{(l)})$$

First, the representation of neighboring nodes near each node is aggregated into a single vector. The current representation of the node is then contacted with the aggregated neighborhood vector. To use the representations in the next step, the concatenated vector is passed through a fully connected layer with σ (nonlinear activation function).

3.5. Imbalanced data classification

An imbalanced dataset refers to a dataset in which the number of instances in different classes varies greatly. A major problem in data mining and machine learning is imbalanced data classification. In the dataset, minority class is a class with less data, and majority class is a class with more data. Ordinary classification algorithms do not perform well in imbalanced datasets; Because the Ordinary classification algorithms tend to majority class training samples. This increases the error in identifying minority instances [53]. Synthetic minority over-sampling technique (SMOTE) is one of the methods that is used to solve the problem of the imbalanced datasets in machine learning [54]. In this method, new samples of the minority class are synthesized to balance the dataset by re-sampling the instances of the minority class.

4. The Proposed Method

In this work, a node classification algorithm is used to classify graph nodes into buggy and bug-free nodes. The general stages of the proposed method are presented in Fig. 2. First, each source code in the database is mapped to an AST-based graph. Then, using the Gumtree algorithm, all graph nodes are labeled with a buggy or bug-free label. Next, a node classifier based on extracted graphs and labeled nodes is trained. Since the data are imbalanced, we use an over-sampling method on the graph data structure. In what follows, we describe the details of converting source code to graphs, labeling nodes using Gumtree, training and testing steps of the proposed method. The training phase includes over-sampling and node classification by using GraphSAGE.

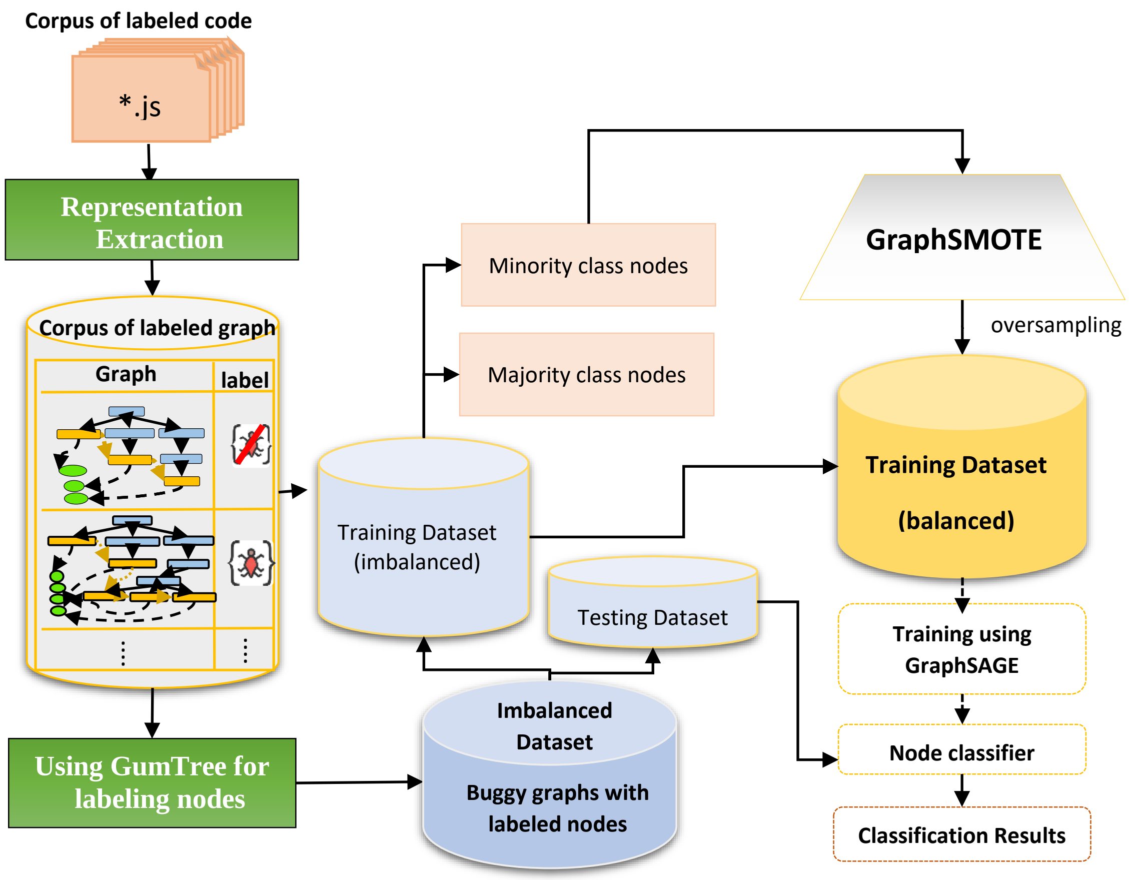
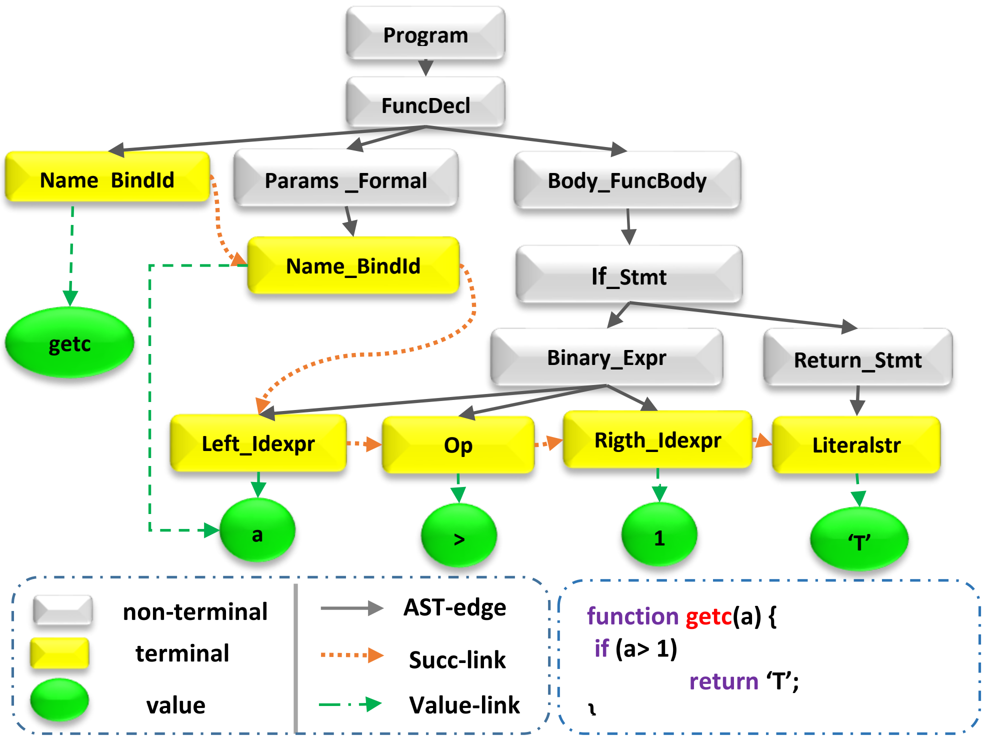


Fig. 2. An overview of the proposed approach

In this paper, a graph-based approach that produces symbolic representations of the code is presented. In this production, the combination of ASTs with additional edges is used, which leading to providing data flow and control flow [8, 55]. In the mapping stage, different types of edges are used to model the syntactic relationships between different tokens. Fig. 3 shows a mapped example of a source code. As we can see, syntax nodes are labeled *non-terminal* based on the program grammar, while syntax tokens are labeled with the string they represent. *AST-edge* edges are used to connect the nodes according to AST [8, 9].



```
function getc(a) {
 if (a> 1)
        return 'T';
}
```

Fig. 3. An example of a program code and its corresponding graph

After the mapping step, we used the Gumtree algorithm [19] to label the graph nodes. Each sample is labeled with one of the buggy and bug-free labels. The architecture of the Gumtree algorithm used in our work is shown in Fig. 4. First, input files buggy *.js and corresponding bug-free *.js are mapped into two graphs, buggy *. json file and corresponding bug-free *. json file. In this stage, the corresponding ASTs and a parser are used. Afterward, these graphs are given to an abstract mapping module to calculate a set of mappings as output. Finally, the output (a set of mappings) is given to an action module to calculate the actual editing script. For our purpose, actions include updating, inserting and deleting. To generate the final output, an abstract output module calculates the output based on input files, graphs, mappings, and editing scripts. In our work, the generated editing script and the final output of the Gumtree algorithm are used to label the graph nodes. For each node in the graph, if an editing script is generated that contains one or more actions, we label that node as buggy and for the other nodes we consider the bug-free label.

In the last stage of our proposed approach, we apply supervised node classification. Nodes in a graph have a type feature. The range of values for this feature includes non-terminal nodes (N), terminal (T) and value nodes (V). In addition to this feature, syntactic features are valuable for extracting useful information from code written in a programming language. In fact, different types of syntax nodes and their relationships are the most efficient features for predicting buggy nodes. Indeed, the various types of syntactic nodes and their relationships are the most efficient features for predicting buggy nodes of the graph. We also encode

$x_s \in R^n$ feature to gather more syntactic AST information for each node in the graph. x_s is the value of a node in a set of syntactic variables.

As mentioned earlier, most lines in the buggy source code are bug-free, and bugs make up only a small part of the program source code. For this reason, we have an imbalanced dataset. To tackle this problem, GraphSMOTE approach [56] is used to balance the training set. GraphSMOTE is a new framework that operates on graph and dose the over-sampling task. This framework uses interpolation in embedded space obtained by a GNN-based feature extractor to generate synthetics minority nodes. It predicts connections for synthetic nodes by using an edge generator to.

Now, we can train a classifier on the new training set. The aim is predicting the category of a node in a binary classification setup, where making it a node classification task. A two-layer GraphSAGE with feature matrix X and adjacency matrix A is intended to train a model. The forward propagation of our model is defined as following form:

$$
\begin{aligned}
H_i^{(2)} &= W^{(2)} \cdot Concat(H_i^{(1)}.H_{M(i)}^{(2)}) \\
H_i^{(1)} &= W^{(1)} \cdot Concat(H_i^{(0)}.H_{M(i)}^{(1)}) \\
H_{M(i)}^{(2)} &= aggregate\left(H_j^{(1)}, \forall \text{ j} \in \text{M(i)}\right) \\
H_{M(i)}^{(1)} &= aggregate\left(H_j^{(0)}, \forall \text{ j} \in \text{M(i)}\right) \\
H_i^{(0)} &= X_i
\end{aligned}
\tag{4}
$$

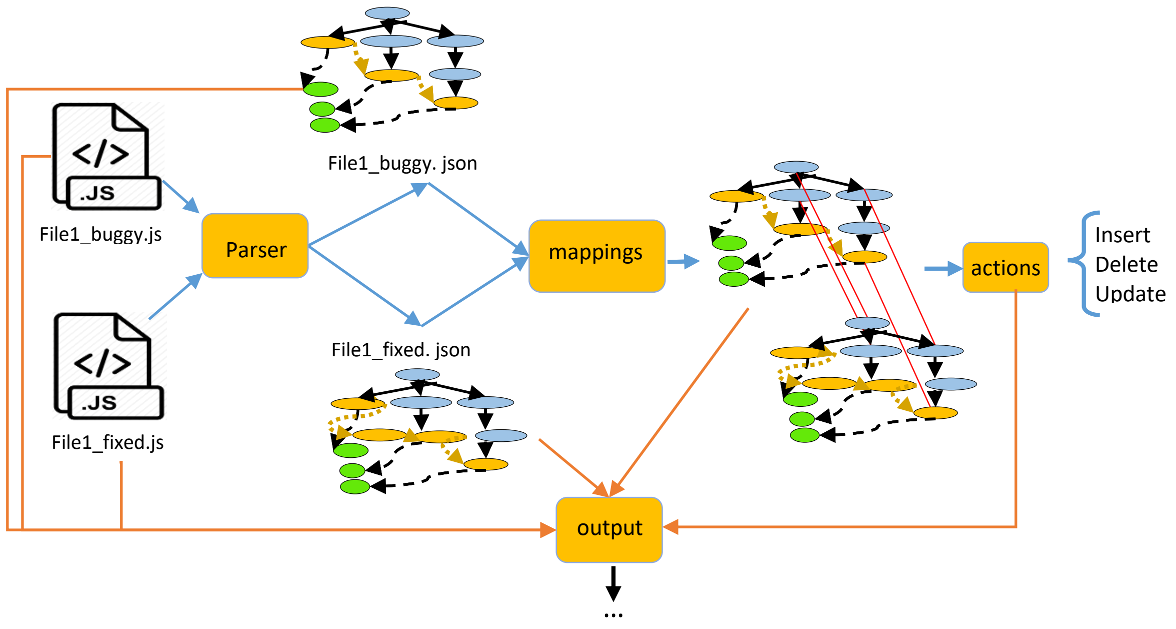


Fig. 4. The architecture of Gumtree algorithm [19]

5. Experimental Setup

5.1. Dataset

We used Hoppity's dataset [8, 57] in the experiments. This dataset contains a lot of samples collected from JavaScript programs in GitHub. In GitHub's commits, various types of changes exist, such as refactorings, bug fixes, feature additions, etc. Only bug fixes are selected based on the number of changes [8]. For a commit, JavaScript files are considered before and after the changes. For our experiments, 27,184 pairs of buggy JavaScript files and their corresponding bug-free files, are selected from this dataset. The total number of nodes is 6,072,325 from which 60% (3,643,395) were used for training, 20% (1,214,465) for validation, and 20% (1,214,465) for testing. In the training set, there are 252,627 buggy nodes and 3,390,768 bug-free nodes. The maximum number of nodes in each graph is set to 800.

5.2. Evaluation Metrics

In the imbalanced dataset, the classification evaluation criterion is different from ordinary datasets. In this dataset, common criteria such as the accuracy criterion alone cannot be used. Therefore, to evaluate our proposed method, other evaluation criteria along with this criterion are used. We measured the performance of our model by the following metrics: Confusion Matrix (CM), accuracy, Precision, Recall, and F1 scores. Confusion Matrix is a suitable metric for measuring the success and efficiency of classification systems. CM_{ij} indicates how many elements of class i are labeled as members of class j. Table 1 shows this matrix. The formula of accuracy and other metrics based on this matrix are shown in Table 2Table 2. Evaluation metrics.

Table 1.Confusion Matrix

	`	*Predicted class*	
		Buggy	**Bug-free**
Actual class	**Buggy**	TP	FN
	Bug-free	FP	TN

Table 2. Evaluation metrics

$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$	$Recall = \frac{TP}{TP + FN}$
$Precision = \frac{TP}{TP + FP}$	$F1 = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}}$

5.3. Experimental Settings

We explore the following questions to analyze the performance of the proposed method:

RQ1: Does the proposed method outperforms other related bug localization methods?

RQ2: What is the effect of oversampling on the performance of the proposed method during node classification?

RQ3: During the training, what is the effect of the parameter *l* (number of GraphSAGE layers) on the performance of the proposed method?

RQ4: How robust is the proposed model to the overfitting problem?

For measuring the performance of the proposed method, we perform experiments based on the mentioned dataset in section 5.1.

6. Experimental results

6.1. Results analysis for RQ1

To evaluate the superiority of the bug localization performance of the proposed method, we compare it with two related methods on JavaScript codes, respectively. Methods that have been compared include Hoppity [8] and TAJS [4]. In this study, the performance of the proposed method has been evaluated in comparison to Hoppity using the accuracy metric. In Table 3, the results of comparing the proposed method with the

Hoppity approach are shown. In this table, the accuracy of our approach is higher than Hoppity. We tested less data than Hoppity due to hardware limitations. Because our approach required producing a large number of nodes. Nevertheless, we came to the desired results. For comparing the proposed method to TAJS, we randomly selected 30 buggy codes from our test set to compare our method with TAJS. The reason for choosing this set and performing the test in this way is the impossibility of automatically comparing our entire test set with TAJS. TAJS static analyzer only accepts JavaScript projects that use ES5. TAJS claims to be very good at detecting undefined property bugs, but as shown in Table 4, it has detected only two bugs of undefined property. TAJS cannot also detect functional bugs and refactoring. It ignores internal library analysis and generates a lot of unrelated false warnings due to unsuccessful location. Table 4 presents the list of selected data and test results in the TAJS, Hoppity, and the proposed method. In this table, symbol ✓ indicates that the method correctly identified bug locations in that code, while symbol × indicates that it incorrectly identified bug locations in that code. As can be seen, our proposed method for bug localization is significantly better than other related methods in this comparison. The number of bug files that have two types of undefined feature bugs and functional bugs in the dataset is very high. An undefined property bug indicates that a variable has not been assigned a value or that the variable has not been declared at all. Our proposed method can identify this type of bugs well because among the considered features, we introduced the parent feature of each node, which can be identified by following the sequence of nodes.

We believe that graphs define the syntactic relationship between program components well. Also, different nodes are defined based on language specifications. The proposed method has introduced information and values of vertices as important and distinguishing features. Therefore, the proposed method is able to detect bugs related to program specifications.

Table 3. Comparison purposed method with Hoppity

Method	Dataset	No. Graphs	Max. Graph Size	Acc (%)
Hoppity	Hoppity OneDiff [57]	36,361	500	**35.5**
Proposed method	Hoppity OneDiff [57]	27,184	800	**90.5**

Table 4. Comparison proposed method with TAJS and Hoppity by 30 random testing points

File (GitHub Link)	TAJS	Hoppity	Proposed method
index.js (Link)	×	×	×
convert.js (Link)	×	×	✓
router.js (Link)	×	×	✓
articles.server.route.js (Link)	✓	✓	✓
index.js (Link)	×	×	×
ListAlbums.js (Link)	✓	×	✓
QuizQuestion.js (Link)	×	×	×
order.js (Link)	×	×	✓
crosshairs.js (Link)	×	×	✓
Advisors.js (Link)	×	×	✓
splash.js (Link)	×	×	✓
Container.js (Link)	×	×	×
index.js (Link)	×	✓	✓
z.js (Link)	×	×	×

count.js (Link)	×	×	×
question.js (Link)	×	×	✓
blink.js (Link)	×	✓	✓
display.js (Link)	×	✓	✓
point.js (Link)	×	×	×
getETHFromFaucet.js (Link)	×	×	×
stream_muting.js (Link)	×	×	✓
mana.js (Link)	×	×	×
gather.js (Link)	×	×	✓
index.js (Link)	×	×	×
Form.js (Link)	×	×	×
before_router_match.js (Link)	×	×	✓
index.js (Link)	×	✓	✓
ROT13.js (Link)	×	×	×
help.js (Link)	×	×	✓
CaseDetailsFileTab.js (Link)	×	×	✓

6.2. Results analysis for RQ2

As mentioned, we are dealing with imbalanced data. Therefore, data (graph nodes) balancing is needed. In this case, we have more than one graph, unlike the usual node classification problems. So, the imbalanced rate (im_{ratio}) in each graph is different. We set this variable to the ratio of the number of minority class to the number of majority class nodes. In Fig. 5 the distribution of the training set before and after over-sampling is shown.

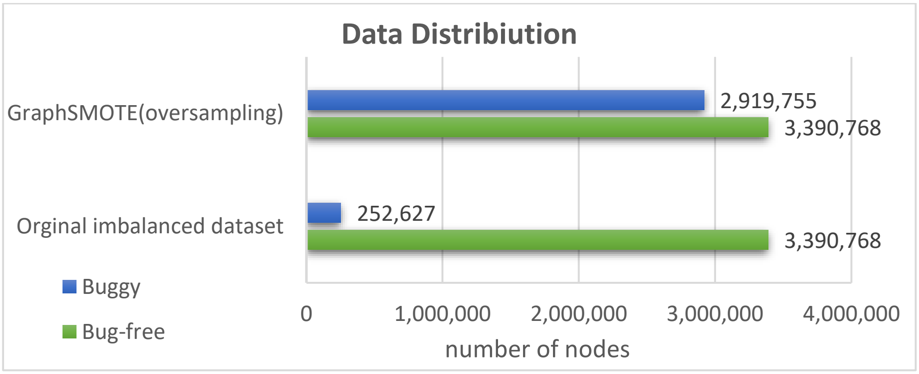
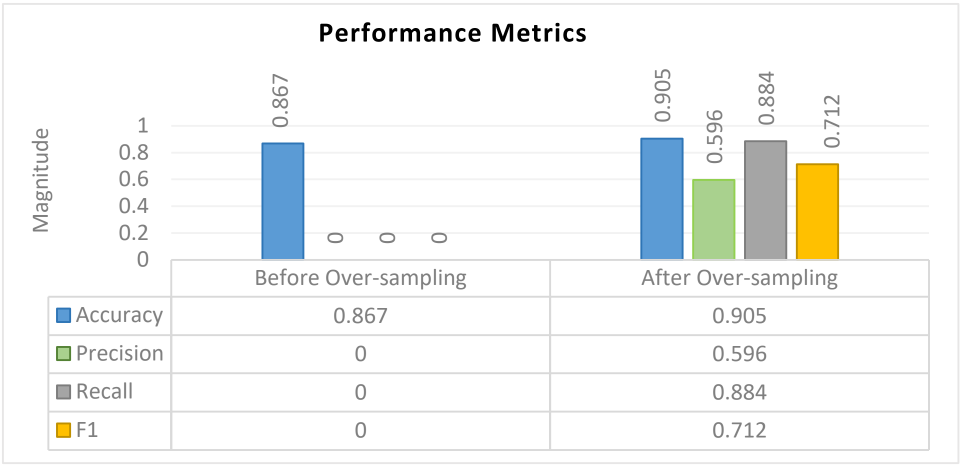


Fig. 5. The distribution of the training set

To study the effect of data balancing, we recorded the results of executing the proposed algorithm on the described dataset before and after the oversampling. The effectiveness of the proposed algorithm is evaluated by the Precision, Recall, F1-scores, and accuracy. Fig. 6 shows the results for this experiment. In this Figure, it can be seen that the accuracy of our proposed method after oversampling is 0.906, and Precision, Recall, and F1 scores are 0.586, 0.880, and 0.703, respectively. High Precision relates to the low false positive rate. We have got .586 Precision which is good. Recall is the ratio of correctly predicted positive observations to the all observations in actual class. We have got recall of 0.880 which is pretty good for this model as it’s above 0.5. F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. We have got 0.703 F1 Score which is a good result. The proposed method after oversampling outperforms the proposed method before oversampling in accuracy, precision, recall and F1. The proposed method before oversampling is good rarely in accuracy metric. Indeed, it classified all nodes of graphs as bug-free nodes.



	Before Over-sampling	After Over-sampling
🟦 Accuracy	0.867	0.905
🟩 Precision	0	0.596
⬜ Recall	0	0.884
🟨 F1	0	0.712

Fig. 6. Overall performance on a held-out set

6.3. Results analysis for RQ3

To investigate the effect of model depth (number of GraphSAGE layers) on classification performance, we change the number of GraphSAGE layers from one to ten in our model. To analyze and obtain the optimized depth, we record recalls, precisions and f1-scores at each stage. In Fig. 7, the results of this experiment are shown. The results show that using 2 GraphSAGE layers brings more accurate results in the mentioned database.

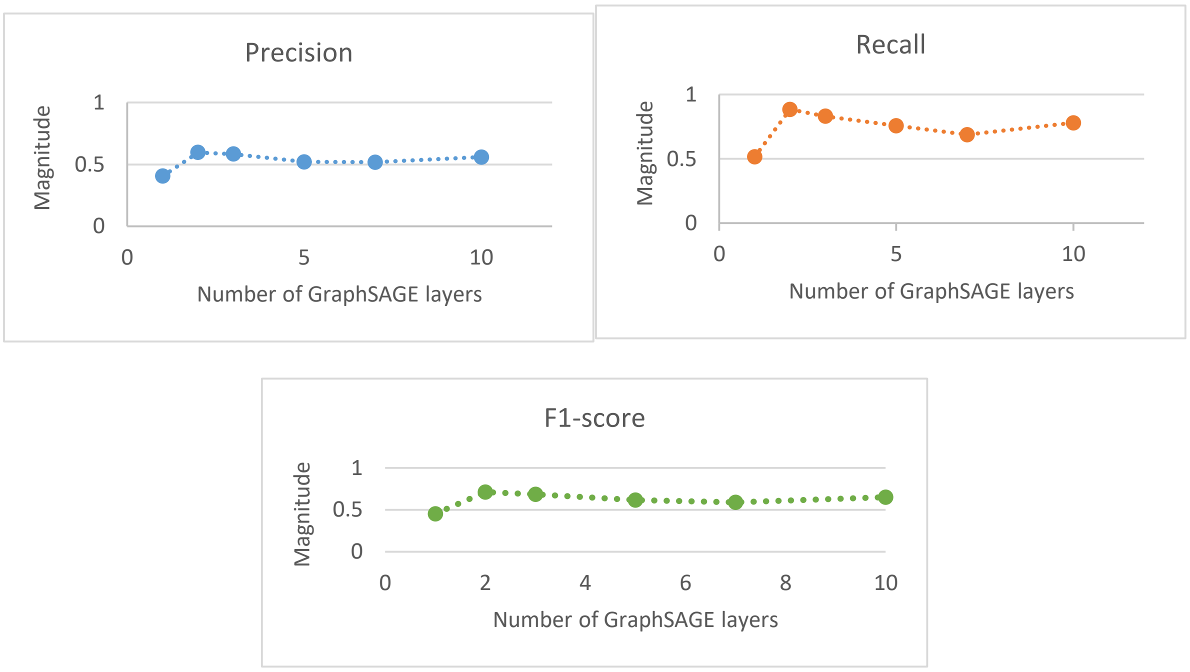


Fig. 7. The effect of model depth on classification performance

The final configuration of our proposed model is shown in Fig. 8. The following hyperparameters provide the best accuracy for this problem. We use two GraphSAGE layers with 32 units for each hidden layer. We use the *mean* aggregator in GraphSAGE layers. A rectified linear units (Relu) layer is used between two GraphSAGE layers. After Relu layer, a dropout layer with rate p=0.5 is used. We used binary cross-entropy as the loss function. The network is trained with the Adam optimizer. Our experiments were implemented on a computer with an Intel(R) Core (TM) i7-7500U3.50GHz CPU, 12GB RAM, and Nvidia 920MX GPU. Python version 3.6.5 was used in the implementation, as well as the Pytorch [58] library (1.7.0), DGL [59] library (0.6.1).

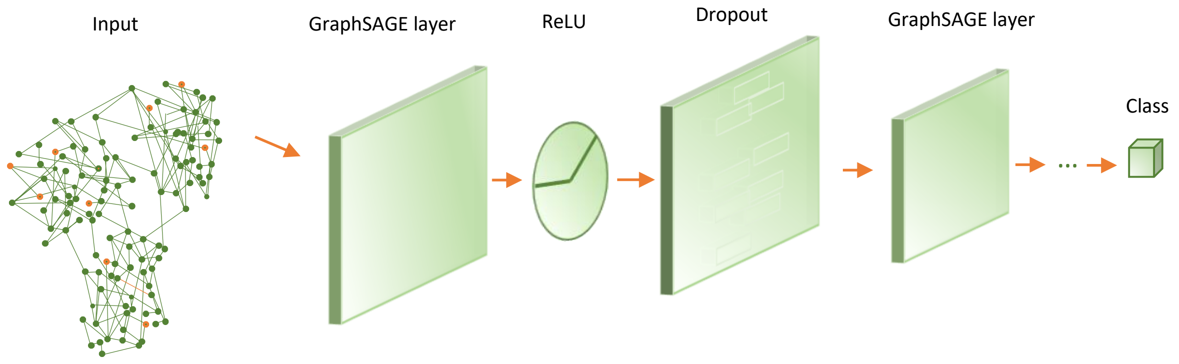
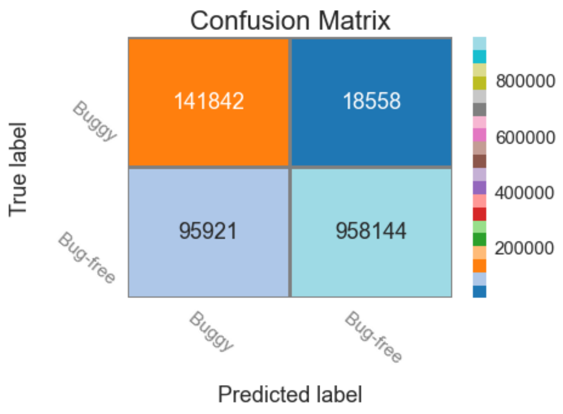


Fig. 8. The configuration of the proposed model

6.4. Results analysis for RQ4

The usual node classification methods perform on the nodes of a single graph, but we deal with a large number of graphs with up to 800 nodes in this problem. To handle the overfitting problem, the dataset is splatted and 20% of the nodes in each graph are selected randomly for testing, 20% for validation, and the remaining for training. Table 5 shows the evaluation results of our model based on the confusion matrix on a held-out test set. Based on this table, the true positive percentage is 0.88, the true negative percentage is 0.90, and the false positive percentage and the false negative percentage are 0.09 and 0.11, respectively. The values of these criteria show that the evaluation results of our approach are promising.

Table 5. Results on true/false predictions



True label \ Predicted label	Buggy	Bug-free
Buggy	141842	18558
Bug-free	95921	958144

7. Conclusion and Future Works

Graphs are rich and flexible structures used to solve many complex problems. Recently, this data structure has been used to model programming source code. The proposed models contain valuable information about the syntactic and semantic structure of programs. In this paper, we proposed a new solution for bug localization that uses graphs and graph-based classifications. We considered the problem of locating the bug as a node classification problem. The results show that the proposed method performs better than other related methods. In the future, one can consider edge classification as well for bug localization.

Declarations

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

Data Availability Statement: Dataset for this research is included in [8].

Funding: The authors received no specific funding for this work.

Author Contributions Statement: The initial idea was from V. R. The design, provision of resources and data collection were performed by L. Y. Data analysis was done by L. Y., S. S., and V. R. The manuscript was written and revised by L. Y. and S. S.

References

[1] I. Vessey, "Expertise in Debugging Computer Programs: A Process Analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, 1985.

[2] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM*, 2004.

[3] "Pmd - an extensible cross-language static code analyzer," https://pmd.github.io/. Accessed:14-09-2018.

[4] S. H. Jensen, A. Møller and P. Thiemann, "Type analysis for javascript," in *Proceedings of the 16th International Symposium on Static Analysis*, 2009.

[5] N. C. Zakas, "ESLint. https://eslint.org/," 2013.

[6] Q. Wang, C. Parnin and A. Orso, "Evaluating the usefulness of IR-based fault localization techniques.," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[7] N. DiGiuseppe and J. A. Jones, "On the influence of multiple faults on coverage-based fault localization," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis(ISSTA)*, 2011.

[8] E. Dinella, H. Dai, Z. Li, M. Naik, . L. Song and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[9] M. Allamanis, M. Brockschmidt and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations(ICLR)*, 2018.

[10] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, p. 61–80, 2008.

[11] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks.," *IEEE Transactions on Neural Networks and Learning Systems (*, pp. 1-21, 2020.

[12] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57-81, 2020.

[13] P. Velickovic, G. Cucurull , . A. Casanova, A. Romero, P. Lio and Y. Bengio, "Graph attention networks," in *ICLR*, 2018.

[14] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *arXiv:1609.02907v4*, 2017.

[15] W. L. Hamilton, R. Ying and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[16] M. Zhang, Z. Cui, M. Neumann and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proceedings of AAAI*, Marion Neumann, and Yixin Chen, 2018.

[17] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[18] "State of the octoverse," 2021. [Online]. Available: https://octoverse.github.com/#top-languages-over-the-years.

[19] J. R. Falleri, F. Morandat, X. Blanc, M. Martinez and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014.

[20] . P. Agarwal and A. Agrawal, "Fault-localization techniques for software systems: A literature review," in *SIGSOFT Software Engineering Notes*, 2014.

[21] W. E. Wong, V. Debroy and B. Choi, "A family of code coveragebased heuristics for effective fault," *The Journal of Systems and Software (JSS)*, vol. 83, no. 2, pp. 188-208, 2010.

[22] B. Korel, "PELAS – Program Error-Locating Assistant System," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, 1988.

[23] H. Agrawal, R. A. De Millo and E. Spafford, "An Execution Backtracking Approach to Program Debugging," *IEEE Software*, vol. 8, no. 5, 1991.

[24] M. Renieris and S. Reiss, "Fault Localization with Nearest Neighbor Queries," in *Proceedings of International Conference on Automated Software Engineering*, 2003.

[25] W. E. Wong, V. Debroy and D. Xu, "Towards better fault localization: a crosstab-based statistical approach," *IEEE Trans*, vol. 42, no. 3, 2012.

[26] . J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *International Conference on Automated Software Engineering (ASE)*, 2005.

[27] C. Le Goues, Automatic program repair using genetic programming, University of Virginia: Ph.D. dissertation, 2013.

[28] . R. Abreu, P. Zoeteweij and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION)*, 2007.

[29] M. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *International Conference on Dependable Systems and Networks (DSN)*, 2002.

[30] L. Gazzola, D. Micucci and L. Mariani, "Automatic Software Repair: A Survey," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 45, no. 1, 2017.

[31] W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," *Int J Soft Eng Knowl Eng*, vol. 19, no. 4, 2009.

[32] D. Hao , T. Xie , L. Zhang , X. Wang, J. Sun and H. Mei, "Test input reduction for result inspection to facilitate fault localization," *Autom Softw Eng*, 2012.

[33] S. K.Lukins, N. A.Kraft and L. H.Etzkorn, "Bug localization using latent Dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, 2012.

[34] S. Wang, D. Lo and J. Lawall, "Compositional vector space models for improved bug localization," in *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[35] D. Kim, Y. Tao, S. Kim and A. Zeller, "Where should we fix this bug? A two-phase recommendation model," *IEEE Trans Softw Eng*, vol. 39, no. 11, 2013.

[36] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *MSR*, 2011.

[37] R. K. Saha, M. Lease, S. Khurshid and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

[38] B. Sisman and A. C. Kak, "Incorporating version histories in information retrieval based bug localization," in *Proceedings of 9th IEEE Working Conference on Mining Software Repositories*, 2012.

[39] W. E. Wong and Y. Qi, "BP Neural Network-based Effective Fault Localization," *International Journal of Software Engineering and Knowledge Engineering,* vol. 19, no. 4, 2019.

[40] L. C. Ascari, L. Y. Araki, A. R. Pozo and S. R. Vergilio, "Exploring Machine Learning Techniques for Fault Localization," in *Proceedings of 10th Latin American Test Workshop*, 2009.

[41] L. Naish, H. Lee and K. Ramamohanarao, "A Model for Spectra-based Software Diagnosis," *Journal of the ACM Transactions on Software Engineering and Methodology,* vol. 20, no. 3, 2011.

[42] C.-C. Lee, P.-C. Chung, J.-R. Tsai and C.-I. Chang, "Robust Radial Basis Function Neural Networks," *IEEE Transactions on Systems,,* vol. 29, no. 6, 1999.

[43] W. Mayer and M. Stumptner, "Model-Based Debugging: State of the Art and Future Challenges," *Electronic Notes in Theoretical Computer Science,* vol. 174, no. 4, 2007.

[44] G. K. Baah, A. Podgurski and M. J. Harrold, "The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis," *IEEE Transactions on Software Engineering,* vol. 36, no. 4, 2010.

[45] R. Abreu and A. J. C. van Gemund, "A Low-cost Approximate Minimal Hitting Set Algorithm and Its Application to Model-based Diagnosis," in *Proceedings of the Eight Symposium on Abstraction, Reformulation, and Approximation,* 2009.

[46] C. Mateis, M. Stumptner and F. Wotawa, "Modeling Java Programs for Diagnosis," in *Proceedings of European Conference on Artificial Intelligence*, 2000.

[47] F. Wotawa, M. Stumptner and W. Mayer, "Model-based Debugging or How to Diagnose Programs Automatically," in *Proceedings of International Conference on Industrial and Engineering, Applications of Artificial Intelligence and Expert Systems*, 2002.

[48] H. Zhong and H. Mei, "Learning a graph-based classifier for fault localization," *Sci China Inf Sci,* vol. 63, 2020.

[49] W. Mayer, M. Stumptner, D. Wieland and F. Wotawa, "Can AI help to Improve Debugging Substantially? Debugging Experiences with Value-based Models," in *Proceedings of European Conference on Artificial Intelligence,* 2002.

[50] W. Mayer and M. Stumptner, "Evaluating Models for Model-Based Debugging," in *Proceedings of ACM International Conference on Automated Software Engineering*, 2008.

[51] O. Vinyals, M. Fortunato and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015.

[52] R. A. Meyers, Encyclopedia of Physical Science and Technology, Third Edition, Academic Press, 2001.

[53] Y. Sun, A. K. C. Wong and M. S. Kamel, "Classification of Imbalancd Data: A Review," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 23, no. 4, 2009.

[54] G. Qiong, W. Xian-Ming, W. Zhao, N. Bing and X. Chun-Sheng, "An Improved SMOTE Algorithm Based on Genetic Algorithm for Imbalanced Data Classification," *Journal of Digital Information Management*, vol. 14, no. 2, 2016.

[55] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," in *Thirty-fifth Annual Conference on Neural Information Processing Systems. NeurIPS*, 2020.

[56] T. Zhao, X. Zhang and S. Wang, "GraphSMOTE: Imbalanced Node Classification on Graphs with Graph Neural Networks," in *Proceedings of the Fourteenth ACM International Conference on Web Search and Data Mining (WSDM '21)*, 2021.

[57] "https://github.com/AI-nstein/hoppity," [Online].

[58] A. Paszke, G. Sam , M. Francisco, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," 2019.

[59] M. Wang, D. Zheng, Z. Ye, Q. Gan, . M. Li, X. Song, J. Zhou, . C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li and Z. Zhang, "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks," in *arXiv:1909.01315*, 2020.