



City Research Online

City St George's, University of London

Citation: Yousofvand, L., Soleimani, S., Rafe, V. & Nikanjam, A. (2026). Graph neural networks for precise bug localization through structural program analysis. *Automated Software Engineering*, 33(1), 17. doi: 10.1007/s10515-025-00556-y

This is the published version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: <https://openaccess.city.ac.uk/id/eprint/36101/>

Link to published version: <https://doi.org/10.1007/s10515-025-00556-y>

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to [City Research Online policy](#).



Graph neural networks for precise bug localization through structural program analysis

Leila Yousofvand¹ · Seyfollah Soleimani² · Vahid Rafe³ · Amin Nikanjam⁴

Received: 2 November 2024 / Accepted: 2 September 2025
© The Author(s) 2025

Abstract

Bug localization (BL) is known as one of the major steps in the program repair process, which generally seeks to find a set of commands causing a program to crash or fail. At the present time, locating bugs and their sources quickly seems to be impossible as the complexity of modern software development and scaling is soaring. Accordingly, there is a huge demand for BL techniques with minimal human intervention. A graph representing source code typically encodes valuable information about both the syntactic and semantic structures of programs. Many software bugs are associated with these structures, making graphs particularly suitable for bug localization (BL). Therefore, the key contributions of this work involve labeling graph nodes, classifying these nodes, and addressing imbalanced classifications within the graph data structure to effectively locate bugs in code. A graph-based bug classifier is initially introduced in the method proposed in this paper. For this purpose, the program source codes are mapped to a graph representation. Since the graph nodes do not have labels, the Guntree algorithm is then exploited to label them by comparing the buggy graphs and the corresponding bug-free ones. Afterward, a trained, supervised node classifier, developed based on a graph neural network (GNN), is applied to classify the nodes into buggy or bug-free ones. Given the imbalance in the data, accuracy, precision, recall, and F1-score metrics are used for evaluation. Experimental results on identical datasets show that the proposed method outperforms other related approaches. The proposed approach effectively localizes a broader spectrum of bug types, such as undefined properties, functional bugs, variable naming errors, and variable misuse issues.

Keywords Bug Localization · Deep Learning · Convolutional Neural Networks · Node Classification

Extended author information available on the last page of the article

1 Introduction

Debuggers consume abundant resources in software development projects. Faults and inefficiencies in debugging techniques also incur billions of dollars in costs for the global economy each year. A software bug is thus an error, defect, or malfunction in a computer program that leads to incorrect, and sometimes, catastrophic results. Bugs can further increase software maintenance expenses. In this line, the program repair process involves loads of operations that can vary depending on the methods and tools applied. In this line, the main debugging operations include bug detection (BD), bug localization (BL), and bug fixing (BF). The BD task accordingly detects whether the input program is buggy. During BL, there are also attempts to find the parts of the program where the bug has occurred. In the BF operation, the corresponding patches or fixes are often generated (Yousfvand et al. 2024). As follows, BL is a vital part of the software debugging process. Obviously, incorrect BL can disrupt the entire location-validation chain, and demands much effort and time to develop a software program. In other words, the bug life cycle in testing is moderated through accurate and efficient BL. As BL is a manual, time-consuming and very expensive task, it often relies on the experience and intuition of software developers to identify and prioritize likely buggy code. This has led to heightened interest in expanding techniques that can help automate partial or complete BL. Automated BL techniques accordingly aid developers modify and reduce search spaces for bug fixes. Therefore, developers need to lay much focus on fewer entities. In this vein, PMD Pmd - an extensible cross-language static code analyzer n.d.), FindBugs Hovemeyer and Pugh (2004), and TAJIS Jensen et al. (2009) have been of widespread use in software debugging as supplementary tools for locating bugs. Even with the success of these techniques, programmers need further improvement. For example, spectral approaches are less efficient in locating multiple bugs than single ones DiGiuseppe and Jones (2011). As another example, information retrieval (IR) methods are significantly less effective when bug reports contain misleading descriptions Wang et al. (2015). For this purpose, much effort has been put into improving BL approaches. From this perspective, a graph-based bug classifier is designed in this paper to use a sample of actual buggy codes as inputs. Of note, actual buggy codes and the corresponding fixed ones are employed since the quality of input influences that of outputs in many classification algorithms.

In computer science, graphs are typically applied to model an assortment of real-world phenomena, such as social networks, semantic structures, and geographical or physical models. Such data structures are also employed to model program source codes. Over the past years, graph-based code representations have been further developed (Dinella et al. 2020; Allamanis et al. 2018), which contain rich information about the semantic and syntactic structures of programs. As is known, many software bugs are related to the semantic and syntactic structures of program commands. Therefore, the input program source code is often mapped to a graph representation for automated BL, and trains a classifier to locate bugs. To perform the classification task, a graph neural network (GNN) is usually implemented (Wu et al. 2020), which helps establish a simple way to predict tasks at the edge, node, and graph levels. It is thus respected as a valuable deep learning (DL) technique. Of note, GNNs are

neural models proposed to summarize and gather information from graph structures, and capture graph dependencies by transmitting messages between collections of objects, such as edges and nodes) (Zhou et al. 2020). Depending on the required type of graphs or data, some GNNs have been further developed, including graph attention networks (GANs) Velickovic et al. (2018) and graph convolutional networks (GCNs). Thanks to the high performance and simplicity of GCNs, they have been thus far employed as the most popular ones to deal with various problems. GCNs have been also very successful in graph topics, such as node ranking, graph classification, and node classification (Yousofvand et al. 2024). For this reason, the GCN model is tapped in the classification step in this paper.

A small training set and an imbalanced dataset are, however, the main challenges in this field. To note, a small training dataset is not sufficient to train a reliable classifier. For example, only 53 bugs had been used for mining in Li and Ernst (2012). To tackle the first challenge, thousands of actual buggy codes and the corresponding bug-free ones are recruited.

In this line, Zhong and Su (2015) had confirmed that bugs could constitute just a tiny part of the program source code, and most lines seemed bug-free. Besides, the training set exhibits a significant class imbalance, where only a small fraction of nodes is labeled as buggy. To handle this challenge, the over-sampling technique is applied to generate more nodes based on minor class data. In this paper, the buggy node refers to a minor class.

As mentioned earlier, the GCN model and the node classification task are proposed in this paper for automated BL purposes. While similar approaches have been successfully applied to statically-typed languages like C#, implementing them in JavaScript poses unique challenges due to the language's dynamic nature and runtime behavior. These differences necessitate more advanced parsing and modeling techniques to accurately represent JavaScript code as a graph structure. In this paper, much focus is laid on JavaScript code, which has been the most popular language on GitHub in the last eight years State of the octoverse (2021). In the proposed method, a graph is thus extracted for each input code, using an abstract syntax tree (AST). The primary contributions accordingly include labeling the training data using the Gumtree algorithm, applying DL for node classification, and handling the issue of imbalanced data in the code graph. Although existing methods are often implemented, the proposed combination occurs for the first time, and it is the initial effort in which such methods are recruited to cope with the problem of BL. In addition, the results show that the given combination outperforms the previous BL methods for JavaScript code in terms of accuracy. An AST differentiation algorithm (namely, a Gumtree algorithm) Falleri et al. (2014) is also exploited to label the graph nodes by comparing the nodes of the buggy graphs with those of the corrected ones. Additionally, various node classification configurations are tested to identify the most effective one.

The remainder of the paper is structured as follows. Section 2 reviews the current state of the art in this field. The required background, such as bugs in JavaScript programs, ASTs, GCNs, and imbalanced datasets are further described in Sect. 3. Section 4 provides a detailed description of the proposed method. Section 5 presents various evaluation metrics and experimental results. Finally, Sect. 6 offers conclusions and discusses future work.

2 Related works

BL techniques have been extensively studied in recent years and are generally categorized into four main categories: spectrum-based, information retrieval (IR)-based, machine learning (ML)-based, model-based and large language model (LLM)-based methods. Each of these paradigms addresses the localization challenge from a different perspective, leveraging distinct types of information such as execution traces, textual similarities, statistical patterns, or formal models. To provide a structured understanding of the field, Table 1 presents a categorized list of representative studies within each approach, while Table 2 offers a comprehensive comparison of these methods with respect to their core mechanisms, data requirements, and bug detection capabilities. These tables also highlight the distinguishing features and advantages of our proposed method, which integrates elements of both ML and model-based strategies to achieve broader bug coverage and enhanced localization precision.

Table 1 Overview of BL technique categories

Category	Description	Key references	Remarks
Spectrum-based	Utilize dynamic information from test executions to identify suspicious program elements based on coverage data and pass/fail outcomes	(Agarwal and Agrawal 2014; Wong et al. 2010; Korel 1988; Agrawal et al. 1991; Renieris and Reiss 2003; Wong et al. 2012; Gazzola et al. 2017; Le Goues 2013; Jones and Harrold 2005, Abreu et al. 2007; Chen et al. 2002; Wong and Qi 2019; Hao et al. 2012)	Effective in many contexts but heavily reliant on test cases. Recent improvements include neural networks and test minimization
IR-based	Treat BL as a document retrieval task, using bug reports as queries and computing similarity with source code	(Rao and Kak 2011; Wanf et al. 2014; Lukins et al. 2012; Kim et al. 2013; Aakanshi et al. 2018, Saha et al. 2013, Sisman and Kak 2012, Wong et al. 2014, Lam et al. 2017, Xu et al. 2025)	Employs methods like LSA, LDA, SVM, and deep learning. Incorporates external sources such as version history and stack traces
ML-based	Leverage labeled datasets (e.g., coverage data, execution outcomes) to train models that predict buggy locations	(Ascari et al. 2009; Naish et al. 2011; Lee et al. 1999; Briand et al. 2007)	Commonly uses BPNNs, SVMs, RBFs, and decision trees. Depends on high-quality feature engineering
Model-based	Compare observed program behavior with a correct reference model to locate discrepancies indicating bugs	(Mayer and Stumptner 2007; Baah et al. 2010; Mateis et al. 2000; Wotawa et al. 2002; Kumari et al. 2019; Zhong and Mei 2020; Mayer et al. 2002; Mayer and Stumptner 2008; Xiao et al. 2019; Vinyals et al. 2015; Yousofvand et al. 2023)	Includes logic- and dependency-based modeling, and recent deep graph-based models. Often limited by domain-specific assumptions
LLM-based	Use pre-trained language models to detect or fix bugs with minimal supervision or data	(Binta Hossain et al. 2024; Do Viet and Markov 2023; Campos 2024)	Includes Codex, CodeBERT, and CodeT5. Promising results but high inference cost and black-box nature are current limitations

Table 2 Comparative analysis

Feature/approach	Spectrum-based	IR-based	ML-based	Model-based	LLM-based	Proposed method
Requires test executions	✓	✗	✓	✓	✗	✗
Utilizes bug reports	✗	✓	✗	✗/Partial	✗	✗
Employs machine learning	✗	✓	✓	✓ (some)	✓	✓
Uses graph representations	✗	✗	✗	✓	✗/Partial	✓ (custom code graphs)
Bug types supported	Limited	Textual	Data-dependent	Domain-specific	Partial	✓ (wide: syntax, semantic, multi-fix)
Handles complex/multi-fix bugs	✗	✗	Partial	Partial	✗/Partial	✓

Our proposed approach is situated at the intersection of ML-, model-, and graph-based techniques, and diverges from traditional BL methods in several keyways:

1. **Wider bug coverage:** The method can detect a broad spectrum of bug types, including misuse of operators or identifiers, undefined property access, incorrect use of `const/let/var`, export statement omissions, variable misnaming or misuse, and more.
2. **Graph-centric representation:** Rather than relying on raw source code or textual similarity, the system utilizes a graph-based representation that encodes semantic and structural relationships in code.
3. **No dependency on test executions or bug reports:** Unlike spectrum-based and IR-based approaches, our technique does not require dynamic execution data or external reports.
4. **Multi-fix capabilities:** The proposed model effectively handles complex bugs that require modifications in multiple code locations, an aspect often overlooked in earlier studies.
5. **Interpretable and lightweight:** In contrast to black-box LLMs, our model operates on explicitly defined graph representations with transparent decision boundaries, facilitating easier debugging and model improvement.

Thus, our method provides a complementary and, in many cases, superior alternative to traditional bug localization pipelines, particularly in scenarios involving structurally complex or semantically rich bug patterns.

3 Preliminaries

In this section, the required background is introduced. The following concepts, namely, the bugs in JavaScript programs, ASTs, GCNs, imbalanced datasets, and the Guntree algorithm are further explained.

3.1 Bug in JavaScript program

JavaScript is a web programming language that is extensively applied in client-side web applications. Despite the incredible popularity of JavaScript, inherent features, such as runtime evaluation and dynamic typing, make it one of the most vulnerable languages. Two examples of JavaScript bugs are shown in Fig. 1, each one with a different type of bug, namely undefined property and functional bug. An undefined property accordingly occurs when a script attempts to access a property of an object that does not exist, and a functional bug represents an error that violates the program specification and yet conforms to the coding rules. In Fig. 1(a) the attribute "value" refers to a tag and the attribute innerHTML refers to the contents between beginning and end of a tag. Using "value" attribute here is wrong and the correct attribute is innerHTML. Using "value", no error is thrown. However, the result will be wrong. In Fig. 1(b) the attribute "floor" should be replaced by "round". The round() function will round a number with a decimal value less than 0.5 down to the nearest lower integer, and round a number with a decimal value of 0.5 or more up to the nearest higher integer. It's a functional bug.

3.2 Graph convolutional network

A graph is a visual depiction of a group of related objects, consisting of a set of vertices (nodes) and edges connecting them. It is denoted as $G=(V, E)$, where $V = \{v_i | i = 1 \dots N\}$ represents the nodes, and $E \subseteq V \times V$ represents the edges between them.

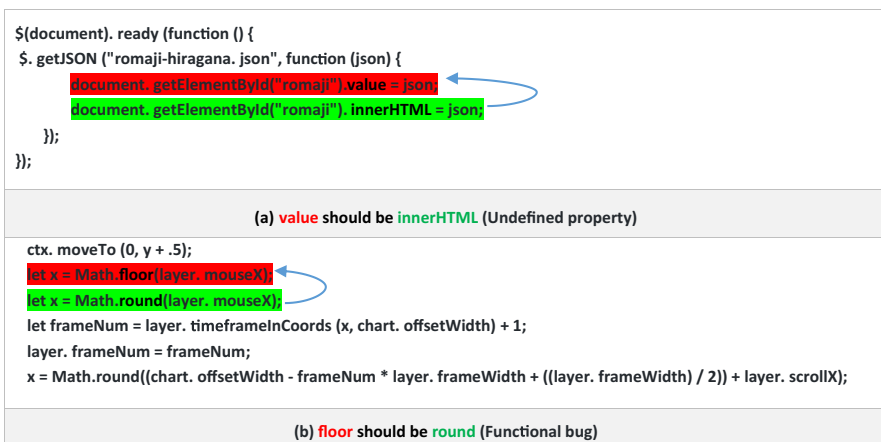


Fig. 1 JavaScript buggy code snippets

The adjacency matrix of the graph is $A \in R^{N \times N}$, while the node feature matrix is $X \in R^{N \times D}$, with N nodes each having D -dimensional feature vectors. $M(i)$ denotes the set of neighbors for node i .

Graph Convolutional Networks (GCNs) are designed for semi-supervised learning on graphs by combining node features with structural information. A graph convolution layer is represented as $H = f(\tilde{D}^{-1} \tilde{A} X W)$, where W is the trainable parameter matrix. Graph convolution involves four steps:

1. Applying a linear transformation to the feature matrix X by multiplying it by W (XW), where the same W is shared across all nodes.
2. Propagating node information to its neighbors using $\tilde{A}Y$, where $Y = XW$, and the propagation to adjacent nodes is given by $(\tilde{A}Y)_i = \sum_j \tilde{A}_{ij} Y_j = Y_i + \sum_{j \in M(i)} Y_j$.
3. Normalizing the propagated information with \tilde{D}^{-1} to maintain a consistent feature scale.
4. Applying a non-linear activation function f .

By stacking multiple graph convolution layers, GCNs capture multi-scale sub-structure features. The GCN propagation rule is $H^{(l+1)} = \sigma(LH^{(l)}W^{(l)})$, where $H^{(l)} = (h_1^l, \dots, h_N^l)^T \in R^{N \times D^l}$ is the node representation in the l -th layer, with $H^{(0)} = X$ as the initial feature matrix. The activation function, such as ReLU, is denoted by $\sigma(\cdot)$. The learned weight matrix in the l -th layer is $W^{(l)} \in R^{D^{(l)} \times D^{(l+1)}}$. The graph Laplacian, used for aggregation, is defined as $L = \hat{A} \triangleq \tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2}$, where $\hat{A} = A + I_N$ and I_N is the identity matrix.

3.3 AST

An AST is a model that characterizes the syntax structure of the source code. It is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the node operators. Thus, the leaves have nullary operators, i.e., variables or constants. It is also similar to the parsing tree, except that it retains the important syntactic structure of the source code while eliminating non-terminals that are not necessary to represent (Meyers 2001). As we know, Parse tree (or concrete syntax tree) is a tree that represents the syntactic structure of a string according to some formal grammar. A program that produces such trees is called a parser.

To extract an AST from a parse tree, the following steps need to be taken:

- Separators and priority markers, such as parentheses, should be removed.
- If parents have only one child, they should be replaced with their child.
- The remaining non-terminals are replaced with operators that are their children.

Figure 2 displays the parse tree for a JavaScript statement `{2 + ""} + 20`; and its corresponding AST. As can be seen, non-terminal nodes (Body, Expression, BinaryExpression, ...) that are not essential are removed, and the non-terminals are then replaced with operators that are their children (For example, `+` replaces BinaryExpression.).

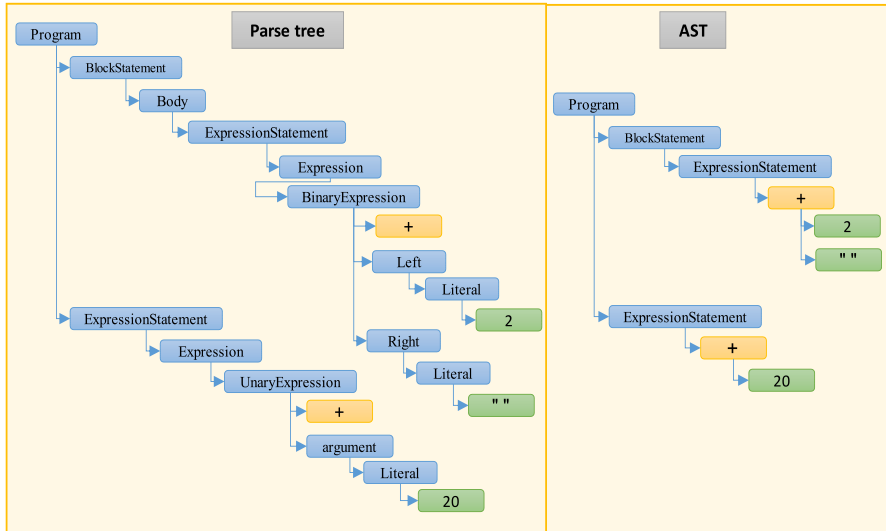


Fig. 2 Parse tree and corresponding abstract syntax tree for a JavaScript statement (`{2+""} + 20;`)

3.4 Imbalanced dataset

As a critical issue, an imbalanced dataset challenges the accuracy of the models managed in classification algorithms. The term imbalanced dataset is generally recruited to refer to a dataset in which the number of instances in different classes greatly varies. The class with fewer data is thus called the minority class, and with more data is named the majority one. In standard classification tasks, class distribution is further considered balanced, and these algorithms do not perform well in imbalanced datasets. The reason is that the common classification algorithms tend to be the majority class, which might increase errors in identifying minority instances (Sun et al. 2009). This is one of the big challenges for imbalanced data classification, which has so far attracted the attention of many experts and researchers in data analysis (Yang and Wu 2006). Numerous methods are also used to deal with the problem of imbalanced datasets in ML, including the data-level approach, in which the distribution of imbalanced class is balanced by re-sampling the data. Under- and over-sampling are also two methods to re-sample imbalanced datasets. In this line, over-sampling seeks to create more samples from the minority class to bring the classes closer together. Under-sampling, however, attempts to sample the majority class. In fact, all the examples in the majority class in this method are not used to bring the ratio of classes closer to each other.

3.5 Gumtree algorithm

In software evolution, a sequence of editing actions, called an editing script, is often applied to the source code (viz., original) to create a new version (namely, modified). The Gumtree algorithm Falleri et al. (2014) refers to a technique for generating editing scripts with regard to the differences in ASTs. The given technique further

produces four types of editing, that is, insert, delete, move, and update. The Gmtree algorithm Falleri et al. (2014) is an AST differentiation algorithm that works in two steps, namely, creating maps, and inferring an editing script.

To compute the mapping between two ASTs, the Gmtree algorithm consists of two successive phases. First, a greedy top-down algorithm is applied to find the isomorphic subtrees of the decreasing height. A mapping is further created between the nodes of these isomorphic subtrees, called anchor mappings. Then, a bottom-up algorithm is utilized in which two nodes are matched if their children (namely, the children of the nodes, and their children, etc.) have many common anchors. Once two nodes match, an optimal algorithm is finally applied to search for additional mappings among their children.

The output of the first step is imported into the algorithm in Chawathe et al. (1996), and the actual editing script is calculated. In this step, a sequence of editing operations is provided that transforms one tree into another. Besides, the four edit operations on the trees are insert, delete, update, and move. An unmapped node in the modified tree is thus considered an insert, and an unmapped node in the original tree is a delete. A pair of mapped nodes with different values is also called an update, and a pair of mapped nodes whose parents are not mapped to each other is a move.

4 Proposed method

A node classification task Kipf and Welling (2017) is performed in this paper for classifying the graph nodes as buggy and bug-free ones. An overview of the proposed method is illustrated in Fig. 3. First, each source code in the database is mapped to a graph representation based on the AST. Afterward, all graph nodes are labeled as buggy or bug-free ones, using the Gmtree algorithm. Next, the training is done. To deal with imbalanced data, over-sampling is finally utilized. The steps in the proposed method are as follows:

- Label all graph nodes in the dataset by the buggy source graph and the corresponding bug-free one, using Algorithm 1.
- Implement Algorithm 2 to balance the dataset generated in the previous step.
- Apply the node classification algorithm on the balanced dataset created in the former step.

In the following, the details of converting the source code into graphs, node labeling using the Gmtree algorithm, and the training and testing steps in the proposed method, are described.

In this paper, a graph-based approach is adopted to create the symbolic representations of the codes. ASTs are further combined with additional edges to provide data-flow and control-flow information Dinella et al. (2020). During this mapping, different types of edges are used to model syntactic relationships between various tokens. In this step, the Hoppity code shared on GitHub (<https://github.com/AI-nst/ein/hoppity>) is utilized. A mapped example of a source code is shown in Fig. 4. As

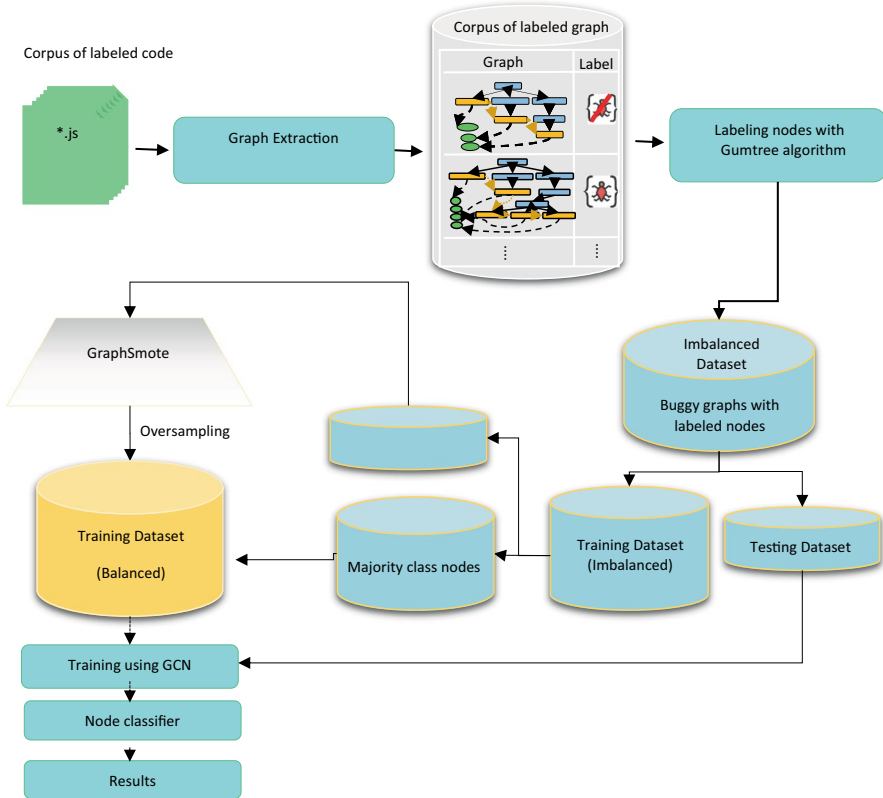


Fig. 3 An overview of the proposed approach

can be seen, the syntax nodes are labeled with non-terminals using the programming language grammar while the syntax tokens are labeled with the string they represent (Dinella et al. (2020); Allamanis et al. 2018).

Upon mapping the code to a graph representation, the graph nodes are labeled by the Gmtree algorithm Falleri et al. (2014). For each sample in this paper, there are two labels, buggy and bug-free. Figure 5 depicts the Gmtree algorithm architecture used for this purpose. As shown, two input files, buggy `*.js` and the corresponding bug-free `*.js`, are mapped to two graphs, with ASTs and a parser. Then, these two graphs are given to an abstract mapping module to calculate a set of mappings as the output. Eventually, the output (namely, a set of mappings) is given to an action module that helps compute the actual editing script. For the problem here, the actions include inserting, deleting, and updating. To generate the final output, an abstract output module further computes the output from the input files, graphs, mappings, and edit script. In this paper, the generated editing script and the final output of the Gmtree algorithm are used for labeling the graph nodes. If an edit script is generated, containing one or more actions, for a node in the graph, that node is labeled as buggy, and a bug-free label is considered for the rest of the nodes.

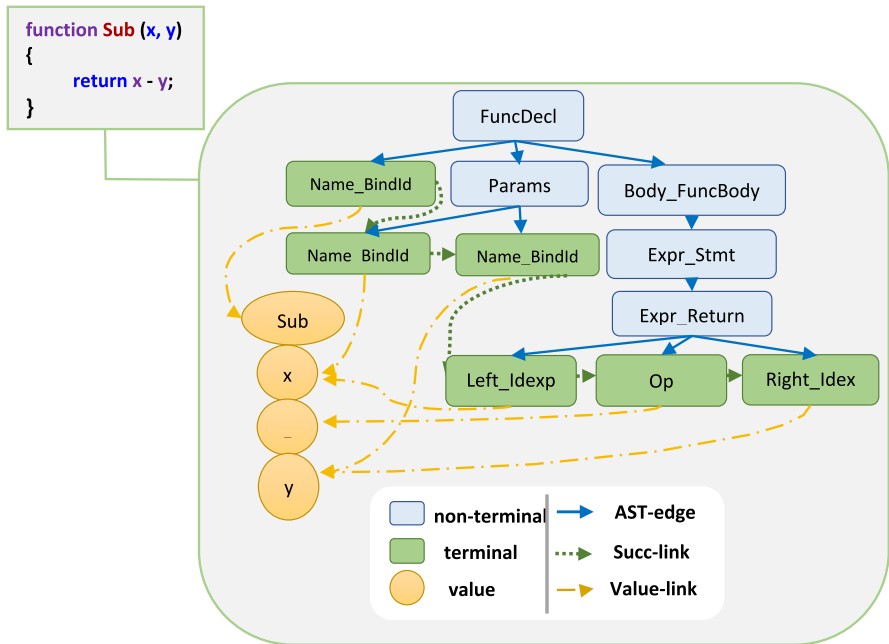


Fig. 4 An example of a program code and its corresponding graph

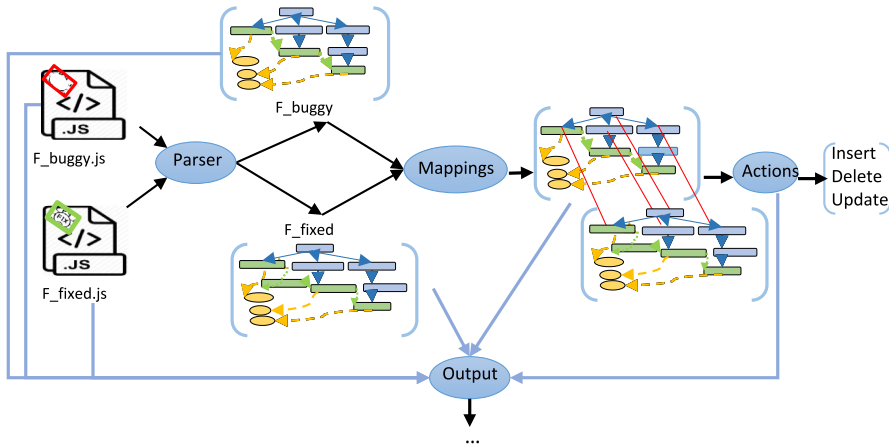


Fig. 5 The architecture of the Gumtree algorithm Falleri et al. (2014)

Algorithm 1 shows the pseudo-code of the labeling algorithm. The input of this algorithm is the graph G_1 , G_2 . G_1 is buggy and G_2 is bug-free. Their nodes are labeled as buggy or bug-free. In line 1, the difference between two graphs G_1 , G_2 is calculated using the Gumtree algorithm and results in a script file. This script file

includes all the operations on the nodes, such as inserting, deleting, updating, etc. In line 2 to 8, labels are generated for each node. In line 3, it is checked whether insert delete or update operations have been created for each node in the editing script produced in the previous step. If it includes this operation, it will be labeled as buggy, otherwise it will be labeled as bug-free.

Algorithm 1 Labeling algorithm

<p>Input: A source graph $G1=(V,E)$ and a destination graph $G2(G1$ is buggy and $G2$ is bug-free)</p> <p>Output: source graph $G1$ which its nodes have buggy or bug-free labels</p> <ol style="list-style-type: none"> 1. text file fl= an editing script generated by the <i>Gumtree</i> algorithm that calculates the difference in AST between $G1$ and $G2$. 2. foreach node v in $G1.V$ do 3. if fl contain {"insert"} or {"delete"} or {"update"} action for node v then 4. $v.label = "buggy"$; 5. else 6. $v.label = "bug-free"$; 7. end if 8. end for 9. return $G1$;
--

After labeling the graph nodes, the supervised node classification task is performed. Graph nodes accordingly encode three different types in a code, including non-terminal (N), terminal (T), and value (V) nodes. Syntactic features are further considered to be important for extracting useful information from the codes written in the programming language. Indeed, various types of syntactic nodes and their relationships are the most efficient features for predicting the buggy nodes of the graph. The feature, $x_s \in R^n$, as the value of a node in a set of syntactic variables (namely, a limited set of non-terminals), is thus encoded to collect more syntactic information from the AST for each node in the graph. All features are syntactic ones, node types, and node labels.

Deciding whether a node is buggy or bug-free is a binary classification problem. As mentioned earlier, bugs make up only a small part of the program source code, and the bulk of lines in the buggy source code are bug-free. Accordingly, an imbalanced training set arises. For this problem, GraphSmote Zhao et al. (2021) is used to balance the training set. This is a novel framework that extends prior over-sampling algorithms to act on graph data. It further generates natural nodes and their related information. The main purpose of this framework is to practice interpolation in an embedding space obtained by a feature extractor based on a GNN to generate synthetic minority nodes. GraphSMOTE also exploits an edge generator to predict connections for synthetic nodes. First, the features of all nodes are extracted using a GNN. Afterwards, oversampling is performed considering a node from the minority (buggy) category and its nearest neighboring node from the same category. In the next step, the representation of a new node is extracted from these two nodes. Then, edges are generated for the new node by a trained edge generator. All the equations related to feature extraction, synthetic node gen-

eration and edge generation are given in Zhao et al. (2021). Algorithm 2 shows the pseudo-code of the over-sampling algorithm. First, the representation of the nodes is obtained using the feature extractor in line 2. Then, over-sampling in the embedding space is performed from line 3 to line 7 to balance the node categories. In line 3, the oversampling rate is determined. This variable is calculated through the ratio of major category samples to the number of minor category samples. In line 4, the minor category is set to “buggy”. In line 7, new artificial nodes are created using the extracted features and variable oversampling rate. Finally, the edges are predicted for the new instances generated in line 8.

Algorithm 2 Oversampling algorithm

<p>Input: An imbalanced dataset of graphs $DG=\{G1,G2,\dots,Gn\}$ Output: A balanced dataset of graphs $DG2$</p> <pre> 1. foreach graph G in DG do 2. Input G to feature extractor; 3. Oversampling-scale=$\ majority\ category\ /\ minority\ category\$; 4. minority category= {"buggy"}; 5. foreach category c in the minority category do 6. foreach i in size(c).oversampling-scale do 7. // "size(c)" denotes the number of nodes belonging to category c in the training graph 8. Generate a new sample (buggy node) v' in category c; //synthetic buggy node generation step 9. Generate A' using an edge generator; //edge generation step, A' is the set of edges for the generated buggy node v' 10. end for; 11. end for; 12. DG2.add(G); 13. end for; 14. return DG2; </pre>
--

Algorithm 3 illustrates the pseudo-code of the node classification task. The input of this algorithm is a large number of graphs. In line 1, all graphs are stored in a large graph Bg . In line 2, the features of the nodes are placed in features. In line 3, the labels of the nodes are placed in labels. In line 4, the nodes in the train set are specified. In line 5 models are made based on GCN. In line 6, the Adam optimizer is applied. In line 7 to 11, there is the prediction operation of nodes. The node classifier can be further trained on the new training set. The forward model is in the following form:

$$H^{(2)} = softmax \left(\hat{A} \left(Relu \left(\hat{A} X W^{(0)} \right) \right) W^{(1)} \right) \quad (1)$$

where $W^{(0)}$ stands for the weight matrix of the input-to-hidden layers, and $W^{(1)}$ denotes the weight matrix of the hidden-to-output ones.

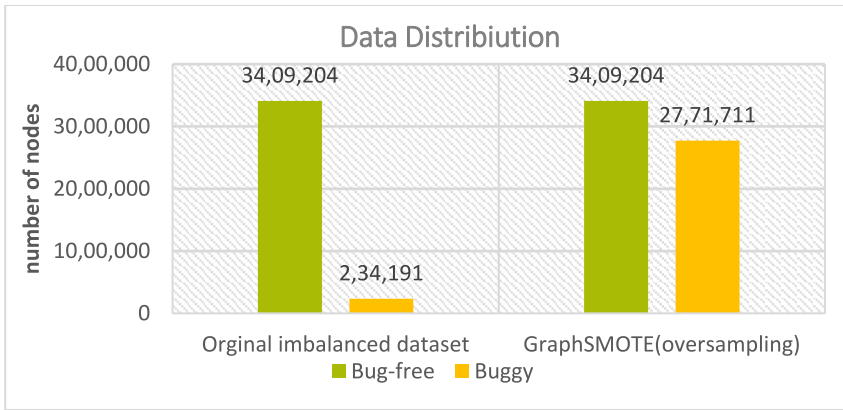


Fig. 6 The distribution of the training set

Unlike the usual node classification problems, there was more than one graph with a different number of nodes, so the imbalanced rate (im_{ratio}) in each graph was different. The value of this variable was further calculated by dividing the number of majority class nodes (N_{major}) by that of the minority ones (N_{minor}), as follows:

$$im_{ratio} = \frac{N_{major}}{N_{minor}} \tag{2}$$

Figure 6 exhibits the distribution of the training set before and after over-sampling.

Several configurations with varying numbers of layers were experimented with, leading to the selection of the following hyperparameters that provided the best accuracy for this task. As shown in Fig. 7, the final model architecture consists of two

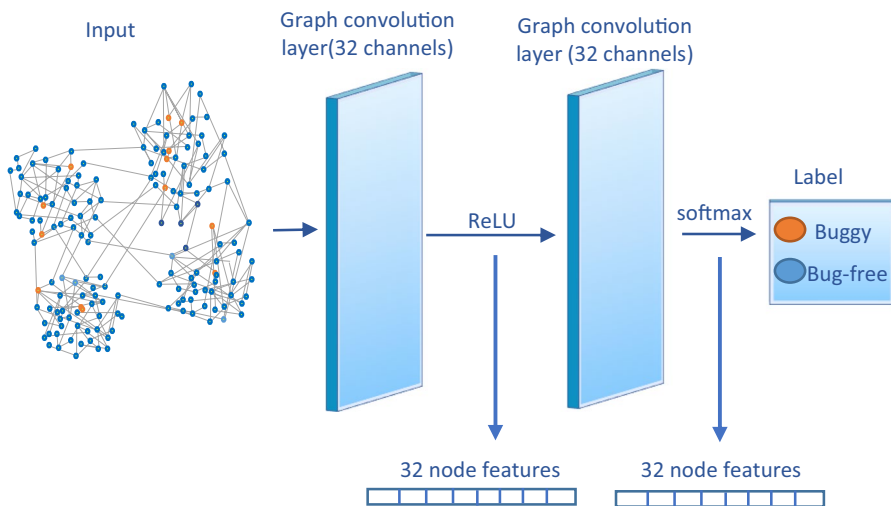


Fig. 7 The configuration of the proposed model

Table 3 Confusion matrix

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

graph convolutional layers, each with 32 hidden units. Each layer computes updated node representations by aggregating information from neighboring nodes. A rectified linear unit (ReLU) activation function is applied after the first graph convolutional layer, followed by the second graph convolutional layer.

The model was trained for 30 epochs using binary cross-entropy as the loss function and optimized with the Adam optimizer. To mitigate overfitting, the dataset was randomly split into training, validation, and test sets.

The experiments were conducted on a machine equipped with an Intel Core i7-7500U 3.50 GHz CPU, 12 GB RAM, and an Nvidia 920MX GPU. The implementation utilized Python 3.6.5, PyTorch 1.7.0, and the Deep Graph Library (DGL) 0.6.1 Wanf et al. (2020).

Unlike common node classification tasks that operate on a single large graph, our approach considers multiple graphs with up to 800 nodes each.

The performance of the given model was further measured by some metrics, including Confusion Matrix (CM) and, accuracy, precision, recall or F1 scores. The CM was thus a suitable measure for assessing the success and efficiency of classification systems. Table 3 shows the main body of CM.

In the case of working with imbalanced datasets, the classification evaluation criterion must also be changed, and ordinary criteria, such as accuracy alone, cannot be used. To evaluate the proposed method, other evaluation criteria were utilized. The formula for accuracy and other metrics is outlined in the following:

$$\begin{aligned}
 \text{accuracy} &= \frac{TP+TN}{TP+TN+FP+FN} \\
 \text{Recall} &= \frac{TP}{TP+FN} \\
 \text{Precision} &= \frac{TP}{TP+FP} \\
 \text{F1} &= \frac{2}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}
 \end{aligned}$$

Figure 8 also illustrates the evaluation results of the model based on the CM on the test data. In this line, the true positive rate (TPR) was 0.81, the true negative rate (TNR) was 0.91, and the false positive rate (FPR) and the false negative rate (FNR) were 0.09 and 0.19, respectively. The values of these criteria revealed that the evaluation results of the approach here were promising.

Table 4 depicts the results for other metrics. Accordingly, the accuracy of the proposed method was 0.901, and the recall, precision, and F1 scores were 0.813, 0.577, and 0.675, respectively. Once again, these results confirmed the appropriateness of the proposed method.

To select the optimal architecture for our task, we experimented with several configurations varying in the number of graph convolutional layers, hidden units per layer, and activation functions. Table 5 summarizes the performance metrics—accuracy, precision, recall, and F1 score—of these configurations.

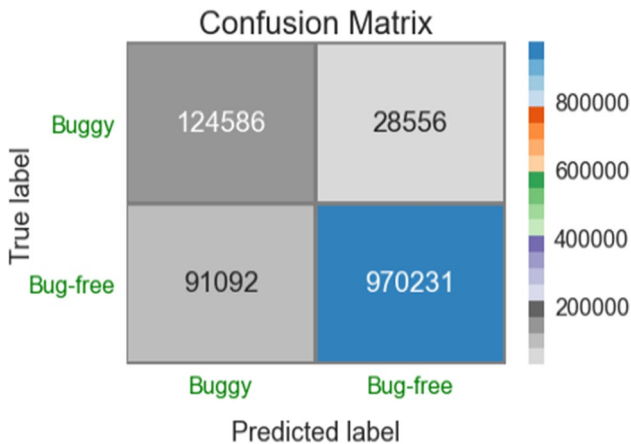


Fig. 8 Results on true/false predictions

Table 4 Overall performance on a holdout set

Accuracy	Precision	Recall	F1
0.901	0.577	0.813	0.675

As shown, the model with two graph convolutional layers, each having 32 hidden units and using ReLU activation, achieved the best overall results, with an accuracy of 0.901, precision of 0.577, recall of 0.813, and F1 score of 0.675. While the configuration with two layers and 64 hidden units achieved a slightly higher recall (0.820), other metrics were lower, confirming that our selected model strikes the best balance among all criteria.

This analysis justifies the choice of the final model architecture presented in this study.

Another metric for evaluating imbalanced datasets was the receiver operating characteristic (ROC) curve, which could help express the TPR against FPR.

The overall accuracy of the test could be higher if the ROC curve was closer to the upper left corner. Of note, the numerical value of the area under the ROC curve (AUC) is clearly a number between 0 and 1, indicating the detection power of a test. If this number is close to one, it means that the data is generally above the bisector

Table 5 Performance comparison of different model configurations

Layers	Hidden units	Activation	Accuracy	Precision	Recall	F1 score
1	32	ReLU	0.880	0.540	0.770	0.640
1	64	ReLU	0.885	0.550	0.775	0.645
1	32	Tanh	0.875	0.535	0.765	0.635
2	32	ReLU	0.901	0.577	0.813	0.675
2	64	ReLU	0.895	0.560	0.820	0.660
2	32	Tanh	0.890	0.555	0.790	0.660
3	32	ReLU	0.890	0.555	0.790	0.660
3	64	ReLU	0.888	0.550	0.785	0.655
3	32	Tanh	0.885	0.545	0.780	0.650

and the true positive rate is high. AUC numbers close to 0.5 show the same equality of correct positive rate and false positive rate, and numbers less than 0.5 indicate higher false positive rate compared to correct positive rate. Figure 9 shows the results of our approach for these two metrics.

To investigate the effect of the model depth on classification performance, the number of different layers in the model here was considered. Figure 10 displays the results of this experiment and confirms the superiority of the chosen model. The results further established that using two convolution graph layers could bring more accurate results in the mentioned database.

The proposed method was further compared with other BL techniques for JavaScript code, including Hoppity Dinella et al. (2020) Yousofvand et.al (2023) and TAJIS Jensen et al. (2009). The results of comparing the proposed method with Hoppity and Yousofvand et.al (2023) are listed in Table 5. As seen, the accuracy of the proposed approach was higher than others. The Hoppity method performed bug fixing in addition to bug localization. Its bug localization operation includes prediction the location of node insertion, update location, and deletion location. Actually, it looks like a 5-classes classification problem. However, in our work and Yousofvand et.al (2023), it is only important to determine the location of the bug. So, it is a two-class classification problem (each node that has the label delete, insert, and update is considered as buggy). Table 3 depicts the localization accuracy of all buggy and bug-free samples. In Table 6, the detection accuracy of buggy samples is calculated by considering two classes for the Hoppity method on the dataset mentioned in Sect. 5. Based on these

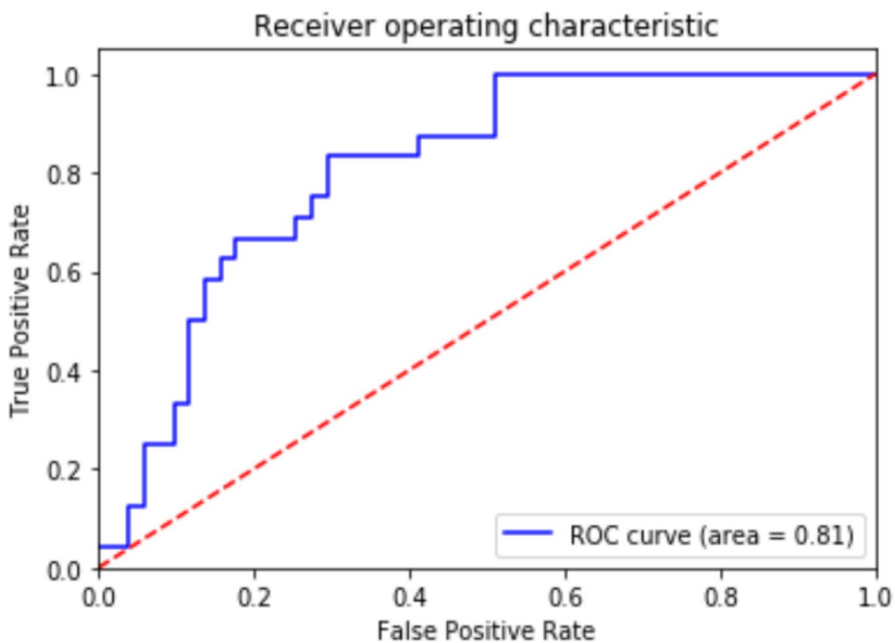


Fig. 9 Receiver operating characteristic (ROC) metric and area under the ROC curve (AUC)

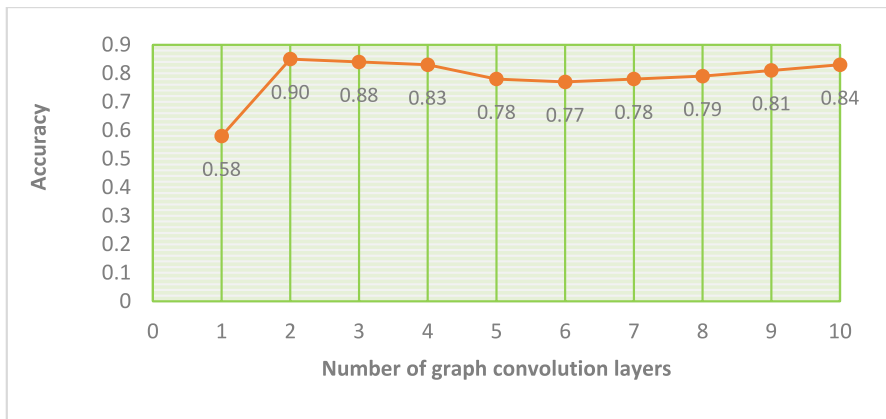


Fig. 10 The effect of model depth on classification performance

Table 6 Comparison proposed method with Hoppity

Method	Dataset	No. Graphs	Max. Graph Size	Acc (%)
Hoppity	Hoppity	27,184	800	62.7
Dinella et al. (2020)	OneDiff https://github.com/AI-nstein/hoppity (n.d.)			
Yousofvand et.al (2023)	Hoppity OneDiff https://github.com/AI-nstein/hoppity (n.d.)	27,184	800	75.8
Proposed method	Hoppity OneDiff https://github.com/AI-nstein/hoppity (n.d.)	27,184	800	90.1

settings, the accuracy of Hoppity method is 62.7% the accuracy of Yousofvand et.al (2023) is 75.8%, and the accuracy of our work is 90.1%.

In total, 30 buggy codes were randomly selected from the test set to compare the proposed method with TAJIS Jensen et al. (2009), because it was impossible to automatically compare the entire test set with the said method. Also, TAJIS static analyzer only accepts JavaScript projects that use ES5. TAJIS further claims to be very good at detecting undefined property bugs, but it detected only two bugs of undefined property type, as shown in Table 7. TAJIS also fails to detect functional bugs and refactorings. Moreover, it ignores internal library analysis and generates many unrelated false warnings due to unsuccessful BL.

In addition to TAJIS and Hoppity, we compared our method with several advanced LLMs, including ChatGPT, CodeT5, and LLaMA 2, to assess their bug detection capabilities. While ChatGPT demonstrated impressive accuracy in identifying syntax-level issues such as typos and missing brackets—consistent with prior findings Mohajr et al. (2024)—its performance dropped significantly on deeper logical bugs and refactoring tasks. Similarly, CodeT5 and LLaMA 2 achieved partial success in detecting certain cases (see Table 7), but neither provided consistent results across bug types. By contrast, our proposed method consistently outperformed these mod-

Table 7 Comparison proposed method with TAJs and Hoppity by 30 random testing points

File (GitHub Link)	Bug type	TAJS	Hoppity	ChatGPT	CodeT5	LLaMA 2	Proposed method
index.js (Link)	undefined property	×	×	×	×	×	×
router.js (Link)	undefined property	×	×	×	×	×	×
convert.js (Link)	undefined property	×	×	×	×	✓	✓
index.js (Link)	undefined property	×	×	✓	×	×	✓
articles.server.route.js (Link)	undefined property	✓	✓	✓	×	✓	✓
QuizQuestion.js (Link)	functional bug	×	×	✓	✓	×	×
ListAlbums.js (Link)	undefined property	✓	×	×	×	×	✓
crosshairs.js (Link)	functional bug	×	×	✓	×	✓	×
order.js (Link)	functional bug	×	×	×	×	×	✓
splash.js (Link)	functional bug	×	×	×	×	×	×
Advisors.js (Link)	functional bug	×	×	×	×	×	✓
index.js (Link)	functional bug	×	✓	×	×	✓	✓
Container.js (Link)	functional bug	×	×	✓	✓	✓	✓
question.js (Link)	functional bug	×	×	×	×	×	×
z.js (Link)	functional bug	×	×	×	×	×	✓
count.js (Link)	functional bug	×	×	×	×	×	×
display.js (Link)	functional bug	×	✓	×	×	×	✓
blink.js (Link)	functional bug	×	✓	×	×	✓	✓
getETHFromFaucet.js (Link)	refactoring	×	×	✓	×	✓	✓
point.js (Link)	refactoring	×	×	×	×	×	×
mana.js (Link)	refactoring	×	×	✓	×	×	✓
stream_muting.js (Link)	refactoring	×	×	✓	✓	✓	×
index.js (Link)	refactoring	×	×	✓	✓	×	×
gather.js (Link)	refactoring	×	×	×	×	✓	✓
before_router_match.js (Link)	refactoring	×	×	✓	×	×	✓
Form.js (Link)	refactoring	×	×	✓	✓	×	✓
ROT13.js (Link)	refactoring	×	×	✓	×	✓	✓
index.js (Link)	refactoring	×	✓	✓	✓	✓	×
CaseDetailsFileTab.js (Link)	refactoring	×	×	✓	×	×	×
help.js (Link)	refactoring	×	×	✓	×	✓	×

els, particularly in functional bug detection and refactoring, owing to its graph-based reasoning over program structure.

For completeness, we report the experimental setup of the LLM-based baselines. For ChatGPT, we used the public May 2024 release (based on GPT-4) via the OpenAI interface. The uniform prompt was: “Does the following code contain any bugs?”,

with raw code snippets directly provided without additional context. For CodeT5 and LLaMA 2, we employed HuggingFace implementations (CodeT5-base and LLaMA-2-7B) with default generation settings (beam search, temperature=0.7). These settings were chosen to ensure fair and reproducible comparisons across all models.

To further validate our method, we additionally examined the quality of node generation for all 30 selected buggy code samples. The purpose of this analysis was not to measure localization accuracy, but to evaluate whether the proposed approach can correctly construct meaningful graph nodes that represent buggy and non-buggy code regions.

As shown in Table 8, the proposed method successfully generated correct graph nodes in 27 out of 30 cases. Only three functional bugs (`splash.js`, `question.js`, and `count.js`) were not properly captured during the node generation phase. These failures can be attributed to the following challenges: (1) functional bugs often involve complex runtime dependencies that are not explicitly encoded in the source-level abstract syntax tree; (2) in such cases, Gumtree's tree-differencing may fail to detect meaningful structural changes between buggy and fixed versions; and (3) the graph abstraction may lack sufficient semantic information to fully represent deeper control-flow or data-flow anomalies.

Overall, the high success rate (90%) in node labeling demonstrates that our proposed approach is capable of producing reliable graph structures for downstream bug localization tasks. The small number of failure cases highlights interesting directions for future work, such as incorporating semantic-aware differencing algorithms or integrating dynamic analysis signals into the graph representation.

We extend our evaluation to include bugs written in Python using the BugsInPy dataset Zhou et al. (2019). This complements the previous analysis that was solely focused on JavaScript-based bugs (from the Hoppity dataset), enabling a broader understanding of performance across languages.

Table 9 presents a comparison between our proposed method and several baseline fault localization techniques—namely MBFL (Metallaxis), SBFL variants (PS, DStar, Ochiai, Tarantula), and ST (Slicing Technique)—across 13 Python projects with a total of 135 real-world bugs.

The results demonstrate that our method achieves the highest total number of successfully localized bugs (59), slightly outperforming all other approaches. In particular:

Our approach consistently outperforms SBFL techniques such as DStar and Tarantula in large projects like `youtube-dl` and `keras`.

Compared to MBFL (Metallaxis), our method localizes more bugs in projects such as `luigi`, `pandas`, and `fastapi`, indicating better adaptability to varied project structures and codebases.

The ST method, while showing strong performance in specific cases (e.g., `the-fuck`), fails to generalize across smaller or more modular projects like `sanic` or `httpie`.

Overall, this additional evaluation confirms that the proposed method maintains its effectiveness across programming languages and different bug datasets, highlighting its potential for wider adoption in real-world debugging scenarios.

Table 8 Evaluation of generated graph nodes for selected buggy codes

File	Bug type	Proposed method (Node Generation)
index.js (Link)	undefined property	✓
router.js (Link)	undefined property	✓
convert.js (Link)	undefined property	✓
index.js (Link)	undefined property	✓
articles.server.route.js (Link)	undefined property	✓
QuizQuestion.js (Link)	functional bug	✓
ListAlbums.js (Link)	undefined property	✓
crosshairs.js (Link)	functional bug	✓
order.js (Link)	functional bug	✓
splash.js (Link)	functional bug	×
Advisors.js (Link)	functional bug	✓
index.js (Link)	functional bug	✓
Container.js (Link)	functional bug	✓
question.js (Link)	functional bug	×
z.js (Link)	functional bug	✓
count.js (Link)	functional bug	×
display.js (Link)	functional bug	✓
blink.js (Link)	functional bug	✓
getETHFromFaucet.js (Link)	refactoring	✓
point.js (Link)	refactoring	✓
mana.js (Link)	refactoring	✓
stream_muting.js (Link)	refactoring	✓
index.js (Link)	refactoring	✓
gather.js (Link)	refactoring	✓
before_router_match.js (Link)	refactoring	✓
Form.js (Link)	refactoring	✓
ROT13.js (Link)	refactoring	✓
index.js (Link)	refactoring	✓
CaseDetailsFileTab.js (Link)	refactoring	✓
help.js (Link)	refactoring	✓

5.1 Failure analysis and the impact of unnatural code on model accuracy

To further evaluate the limitations of the proposed model and understand its behavior in complex scenarios, we conducted a failure analysis focusing on misclassified code samples. Specifically, we examined two main types of errors:

False Positives (FP): Non-buggy nodes that the model incorrectly identified as buggy.

False Negatives (FN): Buggy nodes that the model failed to detect, classifying them as bug-free.

Table 10 presents representative examples of both error types, providing insights into their characteristics and possible underlying causes.

Table 9 Performance comparison of BL techniques on BugsInPy

Project	Bugs	SBFL					ST	Proposed method
		MBFL (Metallaxis)	PS	DStar	Ochiai	Tarantula		
Black	13	5	2	4	4	4	1	4
cookiecutter	4	0	0	2	2	2	0	2
Fastapi	13	3	1	5	5	5	1	6
Httpie	4	0	1	1	1	1	0	1
Keras	18	6	0	6	7	7	0	5
Luigi	13	7	1	5	5	5	2	6
Pandas	18	2	2	3	3	3	0	5
Sanic	3	0	0	1	1	1	0	0
spaCy	6	1	1	3	3	3	0	3
Thefuck	16	7	0	15	15	15	1	14
tornado	4	1	0	2	2	2	0	1
Tqdm	7	1	0	4	4	4	2	4
youtube-dl	16	7	1	6	6	6	1	8
Total	135	40	9	57	58	58	8	59

Table 10 Representative examples of model misclassification cases

Error Type	Example	Description	Probable Cause
False Positive	<pre>function civicaseCaseDetailsFileTabController(\$scope, BulkActions) { \$scope.ts = CRM.ts('civicase'); \$scope.bulkAllowed = BulkActions.areAvailable(); \$scope.bulkAllowed = BulkActions.isAllowed(); }</pre>	This node involves a minor, non-functional change. It was incorrectly flagged as buggy by the model.	High similarity to buggy code patterns in the training data.
False Negative	<pre><main> <section> <p>{this.props.quiz_question.instruction_texts}</p> <p>{this.props.quiz_question.instruction_text}</p> </section> </main></pre>	This node contains a subtle semantic error which was not detected.	The graph structure lacked sufficient semantic cues to reveal the logic flaw.

Our analysis reveals that “unnatural” or rarely occurring code patterns—those that significantly deviate in structure or style from more typical samples—are more prone to misclassification. These edge cases are underrepresented in the training data and often exhibit unusual graph structures, making them difficult for the model to generalize.

This reinforces the importance of training with diverse and semantically rich code samples to enhance the model’s ability to detect bugs in less conventional or rarely encountered scenarios.

6 Threats to validity

We acknowledge several threats to the validity of this study and describe the mitigation strategies employed to address them:

- **Internal Validity:** Model performance may be affected by hyperparameter choices, model architecture, and training procedures. To mitigate this, we followed

established practices such as tuning on a held-out validation set, applying early stopping based on validation loss, and performing grid search over key hyperparameters (e.g., learning rate, hidden size, and number of layers). These steps helped reduce overfitting and ensured robust model selection.

- **Construct Validity:** The labeling of buggy and fixed nodes depends on the Gumtree differencing tool, which may introduce noise. To reduce this risk, we manually inspected a sample of labeled data and filtered out ambiguous diffs. Only changes with clear and consistent structural mappings were retained, minimizing the effect of mislabeling.
- **External Validity:** While the dataset covers a wide range of real-world JavaScript and Python bugs, it may not fully capture the entire space of bugs in practice. Furthermore, the dataset exhibits class imbalance, which could lead to biased learning. To mitigate this, we applied GraphSMOTE, a graph-based oversampling technique designed to generate synthetic samples for underrepresented classes while preserving the topological structure of code graphs. This approach helped the model learn from rare bug patterns and improved its generalization capability.

Despite these precautions, additional evaluations on more diverse datasets and languages (e.g., Java) are planned for future work to further improve generalizability.

7 Conclusion and future works

In this paper, we proposed a novel approach for bug localization (BL) in source code by combining graph-based representations with deep learning techniques. Our method leverages the expressive power of graph structures to capture both syntactic and semantic aspects of program code, which are essential for identifying bugs. Graph Neural Networks (GNNs) were employed to analyze these representations effectively.

The proposed method demonstrated strong performance compared to existing techniques and showed promising capabilities in identifying multiple bugs within a single program. Although the experimental evaluation was conducted on JavaScript programs, the approach is inherently language-agnostic and can be adapted to other programming languages, given appropriate parsing and graph construction mechanisms.

To address the class imbalance, present in the dataset, we applied oversampling strategies. However, the high memory overhead introduced by oversampling presents a limitation. Future work can explore more efficient solutions, such as dynamic sampling or cost-sensitive learning.

Furthermore, we envision extending this work by incorporating gray-box models that combine interpretable white-box classifiers with black-box predictors. This hybrid architecture could leverage automatically generated labels from the black-box component to enhance the white-box model's performance, thereby improving both accuracy and interpretability.

Author contributions It should be declared that Leila Yousefvand, Seifollah Soleimani, Vahid Rafe, and Dr. Amin Nikanjam

- reviewed the manuscript.
- agreed with the content and that all gave explicit consent to submit.
- made substantial contributions to the idea, structure and writing.
- revised the work critically for important intellectual content.
- approved on the version and agreed to be accountable for all aspects of the work.

Funding The authors received no specific funding for this work.

Data availability Dataset for this research is included in Dinella et al. (2020). The code and dataset used in this study are made publicly available to support reproducibility.

No datasets were generated or analysed during the current study.

<https://github.com/leila-you/bug-localization-GCN>

https://drive.google.com/file/d/19Qw_o7pu96jU7XFAEUqEPIV1mkUXBTdM/view?usp=share_link

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Aakanshi, G., Suri, B., Kumar, V., Misra, S., Blažauskas, T., Damaševičius, R.: Software code smell prediction model using Shannon, Rényi and Tsallis entropies. *Entropy* **20**(5), 372 (2018)
- Abreu, R., Zoetewij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Academic and industrial conference practice and research techniques -mutation (Taicpart-Mutation) (2007)
- Agarwal, P., Agrawa, A.: Fault-localization techniques for software systems: a literature review. In: SIGSOFT software engineering notes (2014)
- Agrawal, H., De Millo, R.A., Spafford, E.: An Execution backtracking approach to program debugging. *IEEE Softw.* **8**(5) (1991)
- Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: International conference on learning representations (ICLR) (2018)
- Ascari, L.C., Araki, L.Y., Pozo, A.R., Vergilio, S.R.: Exploring machine learning techniques for fault localization. In: Proceedings of 10th Latin American test workshop (2009)
- Baah, G.K., Podgurski, A., Harrold, M.J.: The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Transactions on Software Engineering* (2010). <https://doi.org/10.1109/TSE.2009.87>
- Binta Hossain, S., Jiang, N., Zhou, Q., Li, X., Chiang, W.-H., Lyu, Y., Nguyen, H., Tripp, O.: A deep dive into large language models for automated bug localization and repair. In: Proceedings of the ACM on software engineering (2024)
- Briand, L.C., Labiche, Y., Liu, X.: Using machine learning to support debugging with tarantula. In: Proceedings of IEEE international symposium on software reliability (2007)

- Campos, V.: Bug detection and localization using pre-trained code language models. In: Informatik 2024, lecture notes in informatics (LNI), Gesellschaft für Informatik (2024)
- Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information: In: Proceedings of the 1996 international conference on management of data (1996)
- Chen, M., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: problem determination in large, dynamic internet services. In: International conference on dependable systems and networks (DSN) (2002)
- DiGiuseppe, N., Jones, J.A.: 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)
- Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., Wang, K.: Hoppity: learning graph transformations to detect and fix bugs in programs. In: International conference on learning representations (ICLR) (2020)
- Do Viet, T., Markov, K.: Using large language models for bug localization and fixing. In: 12th international conference on awareness science and technology (iCAST) (2023)
- Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering (2014)
- Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: a survey. *IEEE Trans. Softw. Eng.* 45(1) (2017)
- Hao, D., Xie, T., Zhang, L., Wang, X., Sun, J., Mei, H.: Test input reduction for result inspection to facilitate fault localization. *Autom. Softw. Eng.* (2012)
- Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: Companion to the conference on object-oriented programming, systems, languages, and applications (OOPSLA). ACM (2004)
- : [Online] (n.d.) Accessed: 17 August 2025 <https://github.com/AI-nstein/hoppity>
- Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javascript. In: Proceedings of the 16th international symposium on static analysis (2009)
- Jones, J.A. Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: International conference on automated software engineering (ASE) (2005)
- Kim, D., Tao, Y., Kim, S., Zeller, A.: Where should we fix this bug? A two-phase recommendation model. *IEEE Trans. Softw. Eng.* 39(11), 1597–1610 (2013)
- Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: The international conference on learning representations (ICLR) (2017)
- Korel, B.: PELAS – program error-locating assistant system. *IEEE Trans. Softw. Eng.* 14(9), 1253–1260 (1988)
- Kumari, M., Misra, A., Misra, S., Fernandez Sanz, L., Damasevicius, R., Singh, V.: Quantitative quality evaluation of software products by considering summary and comments entropy of a reported bug. *Entropy* 21(1), 91 (2019)
- 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) (2017)
- Le Goues, C.: Automatic program repair using genetic programming, University of Virginia: Ph.D. dissertation (2013)
- Lee, C.-C., Chung, P.-C., Tsai, J.-R., Chang, C.-I.: Robust radial basis function neural networks. *IEEE Trans. Syst* 29(6), 674–685 (1999)
- Li, J., Ernst, M.D.: CBCD: cloned buggy code detector. In: Proceedings of the international conference on software engineering (ICSE) (2012)
- Lukins, S.K., Kraft, N.A., Etzkorn, L.H.: Bug localization using latent Dirichlet allocation. *Inf. Softw. Technol.* 52(9) (2012)
- Mateis, C., Stumptner, M., Wotawa, F.: Modeling java programs for diagnosis. In: Proceedings of European conference on artificial intelligence (2000)
- Mayer, W., Stumptner, M.: Evaluating models for model-based debugging. In: Proceedings of ACM international conference on automated software engineering (2008)
- Mayer, W., Stumptner, M., Wieland, D., Wotawa, F.: Can AI help to improve debugging substantially? Debugging experiences with value-based models. In: Proceedings of European conference on artificial intelligence (2002)
- Mayer, W., Stumptner, M.: Model-based debugging: state of the art and future challenges. *Electron. Notes Theor. Comput. Sci.* 174(4), 61–82 (2007)
- Meyers, R.A.: Encyclopedia of physical science and technology, third edition, academic press (2001)

- Mohajer, M.M., Aleithan, R., Harzevili, N.S., Wei, M., Belle, A.B., Viet Pham, H., Wang, S.: Effectiveness of ChatGPT for static analysis: how far are we?. In: Proceedings of the 1st ACM international conference on AI-powered softw (2024)
- Naish, L., Lee, H., Ramamohanarao, K.: A model for spectra-based software diagnosis. *J. ACM Trans. Softw. Eng. Methodol.* **20**(3), 1–32 (2011)
- Pmd - an extensible cross-language static code analyzer.: (n.d.). Accessed 14 Sept 2018 <https://pmd.github.io/>
- Rao, S., Kak, A.: Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th working conference on mining software repositories (MSR) (2011)
- Renieris, M., Reiss, S.: Fault localization with nearest neighbor queries. In: Proceedings of international conference on automated software engineering (2003)
- Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: Proceedings of IEEE/ACM international conference on automated software engineering (ASE) (2013)
- Sisman, B., Kak, A.C.: Incorporating version histories in information retrieval based bug localization. In: Proceedings of 9th IEEE working conference on mining software repositories (2012)
- State of the octoverse.: [Online]. Available: (2021) Accessed 17 August 2025 <https://octoverse.github.com/#top-languages-over-the-years>
- Sun, Y., Wong, A.K.C., Kamel, M.S.: Classification of imbalanced data: a review. *Int. J. Pattern Recognit. Artif. Intell.* **23**(4), 687–719 (2009)
- Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. In: International conference on learning representations (ICLR) (2018)
- Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Advances in neural information processing systems (2015)
- Wang, S., Lo, D., Lawall, J.: Compositional vector space models for improved bug localization. In: Proceedings of IEEE international conference on software maintenance and evolution (ICSME) (2014)
- Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., Zhang, Z.: Deep graph library: a graph-centric, highly-performant package for graph neural networks. Preprint at <https://arxiv.org/abs/1909.01315> (2020)
- Wang, Q., Parnin, C., Orso, A.: Evaluating the usefulness of IR-based fault localization techniques. In: Proceedings of international symposium on software testing and analysis (ISSTA) (2015)
- Wong, W.E., Qi, Y.: BP neural network-based effective fault localization. *Int. J. Softw. Eng. Knowl. Eng.* **19**(4), 573–597 (2019)
- Wong, W.E., Debroy, V., Choi, B.: A family of code coveragebased heuristics for effective fault. *J. Syst. Softw. (JSS)* **83**(2), 188–208 (2010)
- Wong, W.E., Debroy, V., Xu, D.: Towards better fault localization: a crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **42**(3), 378–396 (2012)
- Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H.: Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proceedings of IEEE international conference on software maintenance and evolution (ICSME) (2014)
- Wotawa, F., Stumptner, M., Mayer, W.: 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)
- Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A Comprehensive survey on graph neural networks. In: IEEE transactions on neural networks and learning systems, pp 1–21 (2020)
- Xiao, Y., Keung, J., Bennin, K.E., Mi, Q.: Improving bug localization with word embedding and enhanced convolutional neural networks. *Inf. Softw. Technol.* **105**, 17–29 (2019)
- Xu, H., Wang, Z., Zou, W.: A more accurate bug localization technique for bugs with multiple buggy code files. *Information and Software Technology* **181**, 107675 (2025)
- Yang, Q., Wu, X.: 10 challenging problems in data mining research. *Int. J. Inf. Technol. Decis. Mak.* **5**(04), 597–604 (2006)
- Yousofvand, L., Soleimani, S., Rafe, V.: Automatic bug localization using a combination of deep learning and model transformation through node classification. *Softw. Qual. J.* **31**(4), 1045–1063 (2023)
- Yousofvand, L., Soleimani, S., Rafe, V., Esfandyari, S.: Automatic program bug fixing by focusing on finding the shortest sequence of changes. *Artif. Intell. Rev.* **57**(2), 39 (2024)

- Yousofvand, L., Soleimani, S., Esfandyari, S.: Bug detection using model transformations and deep learning. *Tabriz J. Electr. Eng.* (2024)
- Zhao, T., Zhang, X., Wang, S.: 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)
- Zhong, H., Su, Z.: An empirical study on real bug fixes. In: *Proceedings of the international conference on software engineering (ICSE)* (2015)
- Zhong, H., Mei, H.: Learning a graph-based classifier for fault localization. *Sci. China Inf. Sci.* **63**(6), 162101 (2020)
- Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., Sun, M.: Graph neural networks: a review of methods and applications. *AI Open* **1**, 57–81 (2020)
- Zhou, Y., Liu, S., Wang, Y., Li, Y.: BugsInPy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In: *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Leila Yousofvand¹ · Seyfollah Soleimani² · Vahid Rafe³ · Amin Nikanjam⁴

✉ Vahid Rafe
vahid.rafe@city.ac.uk

¹ Department of Computer Engineering, Faculty of Engineering, Lorestan University, Khorramabad 68135-1911, Iran

² Department of Computer Engineering, Faculty of Engineering, Arak University, 38156-8-8349 Arak, Iran

³ Department of Computer Engineering, City St George's, University of London, London, UK

⁴ Huawei Distributed Scheduling and Data Engine Lab, Toronto, Canada