The Journal of Systems and Software 210 (2024) 111984

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

JITGNN: A deep graph neural network framework for Just-In-Time bug prediction☆

Hossein Keshavarz [a], Gema Rodríguez-Pérez [b,*]

[a] University of Waterloo, Canada
[b] University of British Columbia, Kelowna, Canada

ARTICLE INFO

Keywords:
Just-In-Time bug prediction
Software engineering
Graph neural network
Deep learning

ABSTRACT

Just-In-Time (JIT) bug prediction is the problem of predicting software failure immediately after a change is submitted to the code base. JIT bug prediction is often preferred to other types of bug prediction (subsystem, module, file, class, or function-level) because changes are associated with one developer and the predictions can be applied when the design decisions are fresh in the developer's mind. Many approaches have been proposed to predict correctly whether a software change is bug-inducing. These approaches mainly rely on change metrics such as the size, the number of modified files, and the developer's experience.

Although there has been extensive work on employing deep learning models for other forms of bug prediction, there are few deep models for JIT bug prediction. Furthermore, none of the existing JIT models that use the changed source code consider the graph structure of source codes. In this paper, we propose a JIT model that incorporates both the content and metadata of changes leveraging the graph structure of programs. We designed and built *JITGNN*, a deep graph neural network (GNN) framework for JIT bug prediction. JITGNN uses the abstract syntax trees (ASTs) of changed programs. We evaluate the performance of JITGNN on two datasets and compare it to a baseline and state-of-the-art JIT model. We hypothesize that by including the graph structure of source codes in the JIT bug prediction process, we can improve the performance of the JIT models. Our study, however, shows that JITGNN achieves the same AUC as the state-of-the-art model (JITLine), and they both have the same discriminatory power.

1. Introduction

Software changes such as fixing issues, adding features, and improving existing features are a significant part of the software life cycle. Software evolves because of these changes, but this evolution is not always positive, and these changes may introduce bugs into the system. Ideally, software maintainers want to remove such bugs as soon as possible before users are exposed to them. Bug prediction is studied in different levels of software structure such as subsystems (Nagappan and Ball, 2005; Hassan and Holt, 2005), modules (Graves et al., 2000; Khoshgoftaar and Allen, 2003), and files (Ostrand et al., 2004, 2005; Pan et al., 2006; Moser et al., 2008) to detect and fix bugs before their exposure.

Predicting bugs at the change level (Mockus and Weiss, 2000; Kim et al., 2008) is referred to as *Just-In-Time (JIT)* bug prediction. The advantage of JIT bug prediction (JIT models for short) is that each software change is associated with only one developer, and by predicting bugs immediately after a change is submitted, developers still have the design decisions in mind when alerts are raised. JIT models require less effort than other models to find and fix the bugs (Kamei et al., 2013).

Many JIT models have been proposed (Kamei et al., 2013; Fukushima et al., 2014; Tan et al., 2015; McIntosh and Kamei, 2018; Hoang et al., 2019, 2020; Pornprasit and Tantithamthavorn, 2021). They are commonly machine learning models trained on historical data comprised of software changes with labels identifying whether a software change introduced bugs into the system (bug-inducing) or not (clean). In the context of `git`,[1] we refer to software changes as *commits*. These JIT models are tested on unseen data to predict whether a commit is bug-inducing. This prediction can be either a binary classification (bug-inducing or clean) or a regression problem where the goal is to predict the probability of a commit being bug-inducing.

Early JIT models were logistic regression models trained on a set of features derived from commit metrics (Kamei et al., 2010; Shihab et al.,

☆ Editor: Nicole Novielli.
* Corresponding author.
E-mail addresses: hossein.keshavarz@uwaterloo.ca (H. Keshavarz), gema.rodriguezperez@ubc.ca (G. Rodríguez-Pérez).
[1] https://git-scm.com/

https://doi.org/10.1016/j.jss.2024.111984
Received 14 January 2023; Received in revised form 1 December 2023; Accepted 22 January 2024
Available online 24 January 2024

2012; McIntosh and Kamei, 2018). These metrics include information about different aspects of commits, such as the size, the number of files, directories, and subsystems, the history of the changed files, and the developer's experience but do not include any information about the content of the change. Hence, two arbitrary commits exhibiting comparable metrics will be treated similarly, irrespective of the changes in their source code.

Recent works have studied the inclusion of the syntactic and semantic information of the commits into JIT models (Hoang et al., 2019, 2020; Pornprasit and Tantithamthavorn, 2021). These models show better performances compared to previous models. *JITLine* (Pornprasit and Tantithamthavorn, 2021) is the most recent work in this area, and it outperforms the prior models. However, these approaches consider changes in the source code solely as plain natural language text, disregarding the underlying structure of the program's source code. Source code is not just a sequence of words; it contains meaningful constructs, syntax, and organization that directly impact how a program functions. Thus, by not accounting for the source code structure, these approaches might miss the opportunity to use a comprehensive and accurate understanding of the changes being made in JIT models.

Thus, this paper investigates whether considering the source code structure of programs improves the prediction of JIT models. Our hypothesis is that by adding the content information of the source code the JIT model will have a higher discriminatory power. To test our hypothesis, we built a deep graph neural network (GNN) framework (*JITGNN*) that performs JIT bug prediction by applying graph convolutional networks (GCNs) on the trees obtained from the abstract syntax trees (ASTs) of programs before and after the commit. The use of deep learning on the graph structure of programs derived from their ASTs has gained popularity in different areas of software engineering and programming languages (Alon et al., 2019a,b; Allamanis et al., 2017; Nair et al., 2020; Zhou et al., 2019; Cheng et al., 2021). Some of these works leverage deep GNNs on program graphs and are proven effective.

We formulate the following research questions:

- **RQ1: Can we replicate the baseline and the state-of-the-art JIT models?**
- **RQ2: How does JITGNN perform compared to the baseline and the state-of-the-art JIT models?**
- **RQ3: How does the size of the training set impact the performance of JITGNN on unseen data?**

We used the OpenStack dataset built by McIntosh and Kamei (2018) and the ApacheJIT dataset (Keshavarz and Nagappan, 2022) to answer the above research questions. We compared the performance of JITGNN with two existing JIT models. We used the multiple regression modeling adopted in McIntosh and Kamei (2018) as the baseline model and JITLine (Pornprasit and Tantithamthavorn, 2021) as the state-of-the-art JIT model. We evaluated the performance of the baseline, state-of-the-art, and JITGNN by measuring the area under the ROC curve (AUC), F_1 score , precision, recall, and Matthews Correlation Coefficient (MCC).

Our study shows that (1) using SMOTE to oversample the minority class (bug-inducing) does not improve the binary classification power of JIT models (AUC) but affects the threshold-dependent metrics; (2) the performance of JIT-GNN is comparable to the state-of-the-art and is marginally better than the baseline. Therefore, incorporating the graph structure of source code does not outperform current state-of-the-art JIT models; and (3) increasing the training data size improves the performance of JITGNN.

The organization of this paper is as follows: Section 2, presents the related work. Section 3 introduces the datasets we used in this study. Section 4 explains the architecture and the specifications of JITGNN. Section 5 describes the experiment setups and results. Section 6 presents the results that address our research questions. Section 7 discusses the results, and Section 8 concludes this work.

2. Related work

In this section, we review the literature on (1) bug prediction in general and then give further overviews of the works that are done in (2) JIT bug prediction, (3) deep learning models proposed for this problem, and (4) the use of graph networks to tackle other software engineering problems involving source codes.

2.1. Bug prediction

The problem of bug prediction has been extensively studied in the literature, and many approaches have been proposed to allocate quality assurance resources to the defect-prone entities in software systems. Researchers have worked on various forms of bug prediction. These forms mainly differ in the level of granularity and the definition of a software entity in their contexts. System-level (Nagappan and Ball, 2005; Hassan and Holt, 2005), module-level (Graves et al., 2000; Khoshgoftaar and Allen, 2003), file-level (Ostrand et al., 2004, 2005; Pan et al., 2006; Moser et al., 2008), class-level (El Emam et al., 2001; Pan et al., 2006; Arisholm and Briand, 2006; Rathore and Gupta, 2012), and function-level (Zimmermann and Nagappan, 2008). The features commonly used in these works are code complexity features. However, over time, change process features were proved to be better indicators of defect-prone entities and gained more popularity (Hassan, 2009; Kamei et al., 2010).

Bug prediction can be considered as a regression problem where the goal is to find the number of defects in an entity or to predict the defect density (ratio of defects to size), or a classification problem where the goal is to classify software changes that might be defect-prone.

The first work on change-level bug prediction goes back to Mockus and Weiss's (Mockus and Weiss, 2000). They utilized logistic regression to carry out the classification. The features they selected for their study were a set of code complexity measures such as size, diffusion, files, and author experiences. Then, Kim et al. (2008) classified changes into clean and bug-inducing changes. They selected various features, such as the change message, change's added and deleted lines, source code complexity metrics, and the change metadata (such as time, author, and history), to train an SVM classifier.

2.2. JIT bug prediction

Shihab et al. (2012) conducted an industrial study on risky software changes. They distinguished between bug-inducing and risky changes, which are not necessarily going to introduce bugs into the system but might have a negative impact on the software. They proposed an extensive set of change and code metrics and evaluated the importance of these factors. Kamei et al. (2013) included the effort aspect in the change-level bug prediction and coined the term "Just-In-Time Quality Assurance" or "Just-In-Time Bbg prediction". They proposed a set of 14 change factors in five dimensions that are the most important indicators of defect-proneness of software changes.

Kamei et al.'s work (Kamei et al., 2013) is a turning point in the direction of bug prediction research. Just-In-Time bug prediction attracted much attention after this work, and various aspects of this form of bug prediction have been studied over the years. Fukushima et al. (2014) attempted to solve the problem of small training data by training a cross-project model. They created a pool of training data from 11 projects and selected the same 14 change metrics as Kamei et al. (2013). However, they changed their classifier to the random forest classifier. They showed that models that perform well within a project generally do not work well in the cross-project setting unless the training and test projects are similar. Tan et al. (2015) addressed two problems with the existing JIT bug prediction models by (1) not adopting k-fold cross-validation, and (2) applying four resampling techniques to overcome the class imbalance problem. McIntosh and Kamei (2018)

studied how the evolution of software and software changes impact the performance of JIT models in a longitudinal study.

Recently, a powerful JIT bug prediction model called *JITLine* was proposed (Pornprasit and Tantithamthavorn, 2021). JITLine compares its performance and training time against the existing state-of-the-art on the dataset built by McIntosh and Kamei (2018). JITLine beats the current models in most of the evaluation metrics. Throughout this paper, we interchangeably refer to JITLine as the state-of-the-art or JITLine.

2.3. Deep learning in JIT

Although extensive work has been done on deep learning models in software defect prediction (Wang et al., 2016; Li et al., 2017b; Viet Phan et al., 2017; Dam et al., 2018; Manjula and Florence, 2018; Chen et al., 2019; Wang et al., 2020; Qiao et al., 2020), little research is conducted on deep learning in Just-In-Time bug prediction. The first work in this area is *Deeper* (Yang et al., 2015). Adopting the same 14 change metrics as Kamei et al. (2013), Yang et al. (2015) used deep belief networks (DBN) (Hinton et al., 2006) to map the handcrafted features to feature representations in latent space. Qiao and Wang (2019) proposed a 3-layer neural network that trains upon 10 of the 14 change metrics to predict the probability of a given change-inducing bug in the future. They turned the JIT problem into a ranking problem by dividing the probability of change defectiveness by the size of the change to produce an effort-aware JIT bug prediction (by considering the benefit–cost ratio). Hoang et al. (2019) proposed a model called *DeepJIT*, which, unlike Deeper, is a deep end-to-end model. The deep networks for feature extraction and the deep networks for classification are trained jointly. The authors of DeepJIT further explored the use of deep learning to learn vector representation of code changes in latent space and proposed *CC2Vec* (Hoang et al., 2020). To perform JIT bug prediction, the output vector of CC2Vec is concatenated with the input vectors of DeepJIT (message log and code changes). The resulting vector is given to DeepJIT to conduct the prediction.

2.4. Graph networks in software engineering

Graph neural networks (GNNs) gained substantial popularity over the last decade. In recent years, there has been a trend in software engineering to model programs with graphs and use GNNs to solve software engineering and programming language problems. The gated GNN (GGNN) (Li et al., 2016) is one of the earliest GNNs used in the realm of programs. Li et al. (2016) studied program verification by extending the basic GNN by adding GRU units (Cho et al., 2014) to update the hidden states of the nodes in the graph. Allamanis et al. (2017) use GGNNs to predict variable names and select the correct variables. The authors showed that applying GGNN on the graph model of programs outperforms recurrent neural network (RNN) models built upon the plain code text. Likewise, other areas of software engineering (i.e., program similarity (Nair et al., 2020), software vulnerability (Zhou et al., 2019; Cheng et al., 2021), and code summary (LeClair et al., 2020)) that deal with program source codes started using program graphs and GNNs to consider the graph structure of codes.

To our knowledge, our paper is the first work on JIT bug prediction using graph neural networks (GNNs). There is little work on using program graphs to predict bugs. Zimmermann and Nagappan (2008) employed network analysis on the dependency graph to find the defective binaries in Windows Server 2003. Nadim et al. (2022) use the properties of source code graphs such as density, number of nodes, and number of edges and add them to the common JIT bug prediction metrics for bug prediction. They extract the source code graphs using the XML version of the source code before the change and the one after the change. Bryan and Moriano (2023) create contribution graphs of developers and source codes. They prepare two sets of features. The first is the graph of structural properties that capture the centrality metrics of the bipartite contributions graph. The second is the community and node embeddings obtained using node2vec (Grover and Leskovec, 2016). Then, they use 3 ML methods, namely, logistic regression, random forest, and XGBoost to predict bugs by classifying the edges in the contribution graph.

3. Dataset

This section introduces and describes the JIT bug prediction datasets used in this study.

3.1. OpenStack

Recent JIT models (Hoang et al., 2019, 2020; Pornprasit and Tantithamthavorn, 2021) evaluated their performances on the publicly available dataset collected by McIntosh and Kamei (2018). McIntosh and Kamei (2018) extracted 37,524 change revisions from QT and OpenStack projects for their study. This dataset has 25,150 QT changes mainly written in C/C++ and 12,374 OpenStack changes written in Python.

This dataset uses the commit IDs of the change revisions as unique identifiers. Each revision has a label that identifies whether the change is bug-inducing. The ratio of bug-inducing changes to total changes is 8% and 13% for QT and OpenStack, respectively. In addition to the commit ID and bug-inducing label, the change metrics for all revisions are included in the dataset. The set of change metrics in this dataset combines 14 change metrics used by Kamei et al. (2013) and new change metrics related to the change review process.

In this study, we used the OpenStack subset of this dataset to replicate the baseline and the state-of-the-art JIT models and to evaluate and compare the performance of JITGNN with existing JIT models (RQ1 and RQ2). We did not use the QT data because the number of positive samples in the QT data was too low (8%). However, we did not use the OpenStack data for RQ3 because it is not large enough to perform experiments with various sizes.

3.2. ApacheJIT

The small number of bug-inducing commits is a known limitation of McIntosh and Kamei's dataset (McIntosh and Kamei, 2018) and other available JIT bug prediction datasets. The code review process makes the number of defective accepted changes small. However, even with this process, defective changes are still accepted, and JIT bug prediction attempts to identify these changes. JIT models are mainly machine learning models trained on historical data as seen data to learn features and generalize them to future unseen data. Machine learning models, especially deep ones like JITGNN, are sensitive to small training sets and class imbalance and overfit when the data size is small because they have many parameters. Accordingly, to build an effective JIT model, the model should see many positive samples (i.e., bug-inducing samples) to distinguish between bug-inducing and clean commits.

Available JIT bug prediction datasets do not have many bug-inducing commits, mainly because they are collected from a limited set of projects. The *ApacheJIT* dataset (Keshavarz and Nagappan, 2022) overcomes this problem by building a large cross-project dataset of historical software changes in 14 popular Apache projects. ApacheJIT has 106,674 commits, 28,239 of which are labeled as bug-inducing. Commits in ApacheJIT have the same attributes as the features in Kamei et al. (2013). In this study, we used ApacheJIT for our research questions.

4. JITGNN framework

In this section, we present the *JITGNN* framework. JITGNN is a deep graph neural network model for Just-In-Time bug prediction. Like other

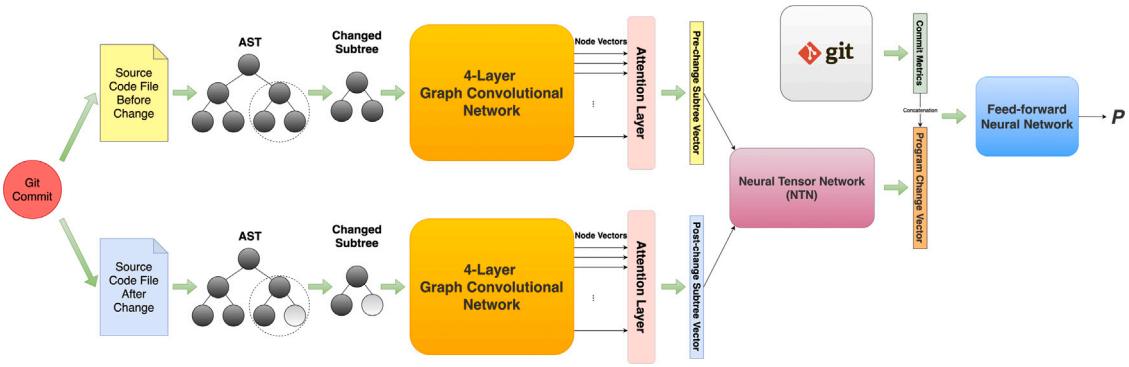


Fig. 1. Overview of the JITGNN workflow. The white node is the changed node identified by GumTreeDiff. In this figure, it is assumed that one file is changed in the commit. Section 4.2 explains how commits with multiple changed files are managed in JITGNN.

JIT models, given a commit, JITGNN assesses the changes that were made in the commit to inform the commit author of the likelihood of introducing a bug into the software with that commit. JITGNN makes this prediction using both the content of the change (changed source code) and the change metadata (commit metrics). The implementation of JITGNN is publicly available both on GitHub[2] and a permanent repository (OSF).[3]

4.1. Overview

To include the syntactic and semantic information of the changed code in its assessment, JITGNN uses graphs to model source codes that are changed in a given commit. JITGNN uses ASTs to represent source codes with graphs. To assess a source code change, for each source code file that is changed in a commit, JITGNN builds one graph from the AST of the source code before the change and one graph from the AST of the source code after the change. Next, JITGNN finds the difference between the two ASTs and extracts the subtrees of the two ASTs that are involved in the change (change subtrees). The change subtree of the source code before the change and the change subtree of the source code after the change are fed into two GNNs followed by attention mechanism layers to obtain two global graph embeddings. The GNNs we employed in JITGNN are graph convolutional networks (GCNs).

The two global graph embeddings are fed into a neural tensor network (NTN) unit, a bilinear neural network, to compute a vector of the relationship between the two embeddings. A vector of commit metrics (listed in Table 1) is attached to the relationship vector to augment the information collected from the content of the change through the networks. The resulting vector is finally given to a feed-forward neural network to predict the probability of the changes in the commit (broadly speaking, the commit itself) introducing bugs into the system. Fig. 1 presents the architecture of JITGNN. JITGNN is an end-to-end model trained jointly, and all its neural components (the two GCNs, the attention layers, the NTN, and the final feed-forward network) are trained simultaneously. One forward pass in the JITGNN training passes all the components mentioned above. At the end of the pass, the output probability is compared with the actual label of the given sample, and the loss is calculated. Using the loss, the derivatives of all parameters in these components are computed, and their parameters are optimized together. Joint training is more effective than training neural parts separately because the parameters are updated with respect to each other, and the final result (Tompson et al., 2014; Qin et al., 2016; Ghasedi Dizaji et al., 2017; Guo et al., 2017).

In the following, we explain this process in detail.

[2] https://github.com/uw-swag/jit-bugpred
[3] https://osf.io/ftgvj/

4.2. Change subtrees & commit graphs

The main part of JITGNN is done by the two GNNs that analyze the content of the changes made in a commit. This part requires the commit to be represented with graphs. A commit can consist of multiple changed files. In this work, we concentrated on the bugs caused by changing source code files rather than all types of files. In JITGNN, source code files are converted to ASTs using their context-free grammar, and the ASTs are used to make the graphs fed into the GNNs.

Depending on the program, the AST of a source code file can be very large, while the commit change in the file often involves a small part of the file. Considering the AST of a source code file before a change and the AST of the same file after the change, the proportion of AST nodes involved in the change to all the nodes is usually small. Hence, analyzing the entire AST is not an ideal choice because the AST has not changed largely, and the information from the small part of the AST that is changed might get lost in the GNN.

To overcome this problem, instead of using the entire AST, we extract subtrees of ASTs that are connected to the changed nodes. To this end, we find the changed nodes between the pre-change AST and the post-change AST using GumTreeDiff (Falleri et al., 2014), a graph differencing tool. These nodes, their parents, their children, and their siblings constitute the AST subtrees that we extract from each AST to feed into the GNNs. We call these AST subtrees *change subtrees*. Each source code changed in a commit has two change subtrees: one *pre-change Subtree* and one *post-change Subtree*.

Each commit in our context consists of multiple source codes, and since we convert source codes to change subtrees, each commit is associated with a set of change subtrees. Keeping change subtrees separated adds hierarchical complexity to the next steps because the ultimate prediction should be conducted on commits, not single source code files. Thus, we combine all the pre-change subtrees of source codes changed in a commit for every commit and form one big disconnected graph called *pre-change commit graph*. Similarly, we make a *post-change commit graph* for post-change subtrees. This does not affect the graphs and only associates each commit with two graphs to simplify the prediction model. Fig. 2 shows an example of how these graphs are formed.

4.3. Node features

Each layer in GNNs passes the features of nodes to the connected nodes and updates their features based on the information received from the neighbor nodes. Therefore, initial node features play crucial roles in GNNs as the final node representations depend on the features with which GNNs start to train. In JITGNN, the initial node features are the combination of *node tokens* and *node change status*:

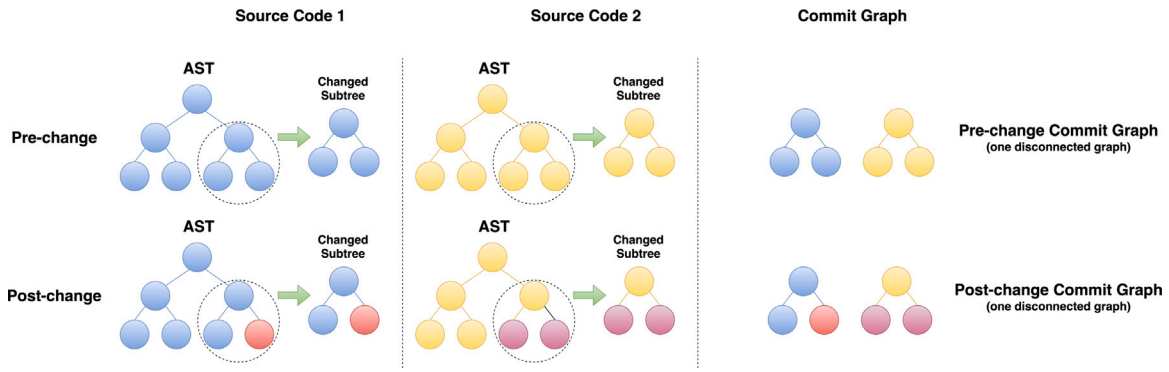


Fig. 2. Commit graphs are formed by making big disconnected graphs from changed subtrees of source codes in commits.

- Nodes in program ASTs have *node type* (grammar non-terminal) and may have *node values* (grammar terminal). In JITGNN, node types (and node values if they exist) form text strings that we refer to as *node tokens*. *NumberLiteral: 5*, *SimpleName: ticket*, *Modifier: static*, and *QualifiedName: TimeUnit.MILLIS* are some Java examples of AST nodes that we extracted.
- The nodes included in the changed subtrees (Section 4.2) are the changed nodes, their parents, their children, and their siblings. Not all the nodes present in change subtrees are changed. We add a feature to the node features obtained from node tokens representing whether or not the node is changed. We call this feature *node change status*, which is 1 if the node is changed and 0 otherwise.

Node tokens are essentially text strings. To feed these attributes to GNNs, they should be converted to numerical vectors. Vectorizing program tokens has been widely studied in recent years (Raychev et al., 2015; Allamanis et al., 2015; Alon et al., 2019b; Feng et al., 2020). Although these studies present sophisticated techniques to transform code tokens into vectors, in our initial experiments, we observed that they do not positively contribute to the final prediction of JITGNN. However, our models achieve good performances with the simple bag-of-words (BoW) technique. Prior JIT models that require vectorizing textual data (either code or commit message) have also used BoW (Hoang et al., 2019, 2020; Pornprasit and Tantithamthavorn, 2021).

The difference between the performance of simple and complex vectorization techniques in JIT models can be attributed to the nature of the text that should be transformed. Although complex models can capture the semantics of the word tokens, unlike code summarization or method name suggestion tasks, JIT models mainly use syntactical information from the text rather than its semantics. Also, training a deep model using BoW is faster because vectors are sparse, and matrix operations inside GNN can be optimized.

The node features we collected using BoW in JITGNN are made up of 0s and 1s because, as we mentioned the node token format above, node tokens are short and tokens obtained from the tokenizer are not repeated. JITGNN concatenates the vectors generated by BoW for node tokens and the single feature of node change status to make the initial node features.

4.4. Graph convolutional network

After extracting the pre-change subtree and the post-change subtree of a given commit, and creating the initial node features, the data is ready to be fed into GNNs in JITGNN. Over the last decade, many types of GNNs were introduced (Wu et al., 2019). Most of these GNNs have the same workflow while the optimization details vary from one to another. In general, in a GNN, the information (features) from nodes are shared with other nodes in the graph through a message-passing system. Each layer implemented in a GNN works as a timestep. At each timestep, every node transmits its hidden state (embedding) to the neighbor nodes (the nodes to which it has direct edges) and receives the hidden states of the neighbor nodes (the nodes that have direct edges to it). All nodes update their hidden states using the hidden states they have received from the neighbors and their own hidden states.

Formally, in graph $G = (V, E, X)$, V is the set of nodes, E is the set of directed edges, and X is the set of initial node features with dimension d. We denote the initial feature vector of node i by $x_i \in \mathbb{R}^d$. At timestep k, node i transmits information (message) $m_i^k = f(h_i^k) \in \mathbb{R}^{d'}$ (depending on f, $d = d'$ may not hold) to neighbor nodes $n_i = \{v_j | (v_i, v_j) \in E\}$. Here f is often the identity function, but generally, it can be any function, and $h_i^k \in \mathbb{R}^d$ is the hidden state or embedding of node i at timestep k. The hidden states are computed in the following way:

$$h_i^k = \begin{cases} x_i & k = 0 \\ g(\mu_i^{k-1}, h_i^{k-1}) & \text{otherwise} \end{cases} \quad (1)$$

Again g can be any function, and μ_i^{k-1} is the aggregation of messages node i receives from the previous layer (timestep $k-1$). Usually, the aggregation is a summation.

Although the backbone of GNNs is similar, how nodes in the graph are updated at each forward pass (function g) is the root of the difference between different GNNs. For example, in a gated graph neural network (GGNN) (Allamanis et al., 2017), the current state and the summation of received hidden states (messages) update the node embedding using GRU units (Cho et al., 2014). In contrast, in a graph convolutional network (GCN) (Kipf and Welling, 2017) embeddings are updated based on shared filter parameters throughout the graph. In other words, the $g(.,.)$ function in GGNN is $GRU(.,.)$ and in GCN is $W^{k-1}\mu_i^{k-1}$, where W^{k-1} is the weights of layer (timestep) $k-1$.

After doing experiments with GGNN – which is commonly used for different software engineering problems that cope with the ASTs of programs (Li et al., 2016; Allamanis et al., 2017; Brockschmidt, 2020; Yu et al., 2022) – and comparing its performance in JITGNN with GCN, we selected GCN over GGNN because it trains faster and outperformed GGNN in our experiments. JITGNN employs two 4-layer GCNs: one GCN unit for the pre-change subtree and one for the post-change. As we explained above, having four layers (timesteps) in a GCN means that at each forward pass during the training, node hidden states are sent 4 times throughout the graph and nodes update their hidden states four times by the received messages. In other words, the information of node i, at each iteration, directly goes to the neighbor nodes and indirectly goes to the nodes in $r_i = \{v_j | 2 <= d(v_i, v_j) <= 4\}$, where $d(v_i, v_j)$ is the distance between nodes i and j (number of edges between v_i and v_j in the shortest path between the two nodes).

For example, if we have $d(v_i, v_j) = 2$, it means that there is node v_k to which v_i has a direct edge and v_k has a direct edge to v_j. At timestep 1, the information of v_i goes to v_k and the information of v_k goes to v_j. All the nodes update their hidden states with the messages they have received. It means that now, the embedding of v_k incorporates

information from v_i, which is transmitted to v_j at the next timestep. Therefore, at timestep 2, v_j receives information from v_k that includes the information of v_i. We say v_j indirectly has received information from v_i.

In this system, although nodes do not receive any information from the nodes that are at distances greater than four in one iteration, throughout the training and after a certain number of iterations, every node, directly or indirectly, receives information from other nodes. This scenario is ideal because distant nodes should minimally affect each other, while close nodes should greatly impact each other. In the scenario discussed above, we see that the closer the node is, the more information it gets from v_i. For example, if node v_j is a neighbor of node v_i, in each iteration in a 4-layer GCN, v_j directly receives information from v_i 4 times. Respectively, the farther nodes indirectly receive information from v_i fewer times.

Although ASTs – and accordingly change subtrees – are directed graphs, following Kipf and Welling (2017), we made edges undirected in JITGNN. In an undirected graph, the hidden states of connected nodes are transmitted in both directions simultaneously. Additionally, we added self-loop edges so the hidden state of a node can be transmitted to itself because, originally, GCN does not directly use the current hidden state of the node to update itself. Adding self-loop edges keeps the hidden states and adds them to the received messages.

After the training of JITGNN, the nodes in the pre-change and post-change subtrees will have new embeddings (their final hidden states) that are their vector representations in higher dimensions and because JITGNN is trained jointly, the embeddings are adjusted and set in a way that the ultimate predictions would be as accurate as possible. As suggested by Kipf and Welling (2017), we normalized the adjacency matrices of the graphs and the initial node features to train GCNs.

4.5. Attention mechanism

The goal of JITGNN is to make one prediction for each commit that is given to it. Accordingly, the embeddings of the nodes that are produced by GCNs should be aggregated to make one single embedding for both subtrees of each change. This approach makes it feasible to compare the single embedding of the pre-change subtree with the single embedding of the post-change subtree and conduct the prediction.

There are multiple techniques to aggregate the embeddings of the nodes in a graph and make a single global embedding. Commonly, this aggregation is done by taking the average of node embeddings (Wu et al., 2019). In other words, the sum of the values of node embeddings at each dimension divided by the number of nodes gives the values of global graph embedding at the same dimension in this approach. The problem with this technique is that all nodes contribute equally to the representation of the global graph embedding.

Another approach is to use a dummy supernode (Scarselli et al., 2009; Li et al., 2017a). The idea in this technique is to add a virtual node to the graph connected to all the nodes in the graph in the receiving direction (there is no edge from this node to other nodes, while from all other nodes, there is an edge to this node). During the training, this supernode is updated like all other nodes, and because the weights are adjusted during the training, this node contains more information from the nodes that are more important in the graph.

In JITGNN, we implemented an approach based on the attention mechanism to take the weighted average of the nodes as the global graph embedding. The weights assigned to the nodes in this technique are parameters that are trainable by the network, and training the entire framework jointly will adjust these weights in the most optimized way. This attention layer was first used in Nair et al. (2020). Formally we have the following:

$$W_i' = \sigma(h_i.tanh(\frac{1}{|V|}\Sigma_{i\in V} h_i.W_i)), \tag{2}$$

where W_i is the attention layer weight that is corresponding to node i and W_i' is the importance weight assigned to node i. Initially, Ws are randomly set based on Xavier initialization (Glorot and Bengio, 2010). As the attention layer is jointly trained, W's are adjusted to take the weighted average of the node embeddings and set it as the global graph embedding. We call the global graph embeddings obtained from the pre-change subtree and post-change subtree, *Pre-change Subtree Embedding* and *Post-change Subtree Embedding* respectively.

4.6. Neural tensor network

The embedding vectors should be aggregated after obtaining the global embeddings of the pre-change subtree and the post-change subtree from the two GCNs. Aggregating the two vectors and having only one vector is necessary because, in the last step, one vector is given to the feed-forward network to produce a probability. Therefore, the aggregated vector at this step should contain information about the change. We call this vector *Program Change Embedding* because given the pre-change subtree embedding and the post-change subtree embedding, this vector represents the change itself.

The choice of the aggregation function at this step is very critical because, as we mentioned above, the change embedding should represent the change and it majorly contributes to the prediction that is carried out at the final step. Similar to obtaining the global graph embedding, common approaches are pairwise distance measures which are static operations on the pre-change and post-change subtree embeddings (Nikolentzos et al., 2017; Wills and Meyer, 2020; Ma et al., 2021). Like before, we leveraged neural network potentials to make a more dynamic approach that is trainable to achieve the best performance of the model. We implemented a neural network layer that is jointly trained with other JITGNN components and compares the two input vectors. This layer is based on the neural tensor network (NTN) proposed in Socher et al. (2013).

NTN was aimed at finding out whether or not there is a relationship between two entities. NTN replaces the classic linear neural network with a bilinear neural network to capture the bidirectional relationship between the two input vectors:

$$S(e_1, e_2) = f(e_1^T W e_2 + W' \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} + b). \tag{3}$$

In Eq. (3), $S \in \mathbb{R}^k$ is the vector of relationship score between entity vectors $e_1, e_2 \in \mathbb{R}^d$, f is a non-linear function, and $W \in \mathbb{R}^{d\times d\times k}, W' \in \mathbb{R}^{k\times 2d}, b \in \mathbb{R}^k$ are NTN parameters (weights and bias). In this equation, $e_1^T W e_2$ is a bilinear function in which both vectors e_1 and e_2 interact on every slice $i = 1, \ldots, k$ of W's third dimension to generate a vector in $\mathbb{R}^k$. The linear part of this equation ($W' \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} + b$) is the classic neural network that receives the concatenation of e_1 and e_2 and outputs another vector in $\mathbb{R}^k$. The sum of the linear and bilinear parts in S generates a vector given to an activation function to produce the relationship vector S. The activation function f used in JITGNN is *ReLU* (Nair and Hinton, 2010).

4.7. Commit metrics

From the early research on JIT bug prediction, commit metrics have been part of many JIT models up to this point. They have always had a positive effect on the performance of the prediction. Our JITGNN experiments with and without commit metrics were aligned with this observation; therefore, we included the commit metrics in JITGNN.

We adopted the commit metrics used by Kamei et al. (2013). Table 1 shows the list and the description of these metrics. These metrics are categorized into four classes: *Size*, *Diffusion*, *History*, *Experience*. In JITGNN, each commit metric is considered a feature, and the commit metrics form a feature vector for each commit. This vector is concatenated with the relationship vector obtained from the NTN module in the previous step. The commit metric vectors are normalized before concatenation to have the same scale as the relationship vectors. The

Table 1
Commit metrics in JITGNN.

Class	Metric	Description
Size	la	total number of lines added in commit
	ld	total number of lines deleted in commit
Diffusion	nf	number of files modified in commit
	nd	number of directories modified in commit
	ns	number of subsystems modified in commit
	ent	distribution of change over files in commit
History	ndev	number of unique developers changed modified files
	age	average time from the previous change of modified files
	nuc	number of unique changes happened to modified files
Experience	aexp	number of commit author's prior changes
	arexp	commit author recent experience
	asexp	number of commit author's prior changes in subsystem

combination of the commit metrics and the relationship vector serves as the ultimate change vector of the commit because it contains information about the change metadata (commit metrics) and the change content (the vector of the relationship between the pre-change subtree and the post-change subtree).

4.8. Feed-forward neural network

In the last step, the ultimate change vector generated by combining the commit metrics and the relationship vector produces the probability of the given commit being bug-inducing. We implemented a feed-forward neural network at the end of the JITGNN framework to receive the change vector and apply the function $f(W^T X+b)$, where f is a non-linear activation function, $W \in \mathbb{R}^d$ is the network weight parameter, $X \in \mathbb{R}^d$ is the change embedding vector, and $b \in \mathbb{R}$ is the network bias parameter. The activation function we used in JITGNN is the *Sigmoid* function because we wanted JITGNN to output a probability rather than classify the commit.

Training JITGNN is supervised, and the loss of each forward pass in JITGNN will be computed based on the actual label of the training sample. Concretely, each training sample takes all the steps discussed above, and finally, JITGNN outputs a probability value indicating how likely the sample is to be a bug-inducing commit. The generated probability, along with the actual label of the sample are given to the *Binary Cross-Entropy (BCE)* function to compute the loss for the sample. The higher the generated probability, the lower the loss for actual bug-inducing commits. On the other hand, for clean commits, a high probability produces a high loss. After obtaining the sample loss, a backpropagation pass is performed to update the model's parameters based on the loss value. In JITGNN, the parameters are all the parameters in the two GCNs, the attention layers, the NTN module, and the final feed-forward neural network. The backpropagation updates (optimizes) the parameters to reduce the loss in future iterations. The optimizer we used in JITGNN is *Adam* (Kingma and Ba, 2015).

5. Experiment design

In this section, we explain the design of our experiment to answer our three research questions.

5.1. Data preparation

As we explained in Section 3, we used the OpenStack and ApacheJIT datasets. We take the following steps to prepare OpenStack and ApacheJIT datasets for training JITGNN.

5.1.1. JITGNN compatibility

The commits given to JITGNN must include at least one program source code. Moreover, JITGNN works with the version of the source code before the change and the version after the change. Therefore, the source codes must be modified. Deleted source codes or newly added ones should not be included in the data as these cases only have one version.

The commits in ApacheJIT already comply with the two rules mentioned above. From the OpenStack dataset, we filtered out the commits in which all the files are non-source-code or all source code lines are deleted or newly added. We refer to this version of the OpenStack dataset as *OpenStack v2*. This dataset has 10,196 commits, of which 1,474 are bug-inducing (Table 2). For RQ1, we used the original OpenStack dataset to replicate the results of existing JIT models. In RQ2, to compare models fairly, we replaced the original OpenStack with OpenStack v2.

5.1.2. Training–test split

To split the data into training and test sets, we considered the real-life scenario and attempted to recreate it as much as possible. In practice, JIT models should notify developers of the bug-inducing likelihood of the changes they have recently submitted. In this scenario, JIT models trained on previous data should effectively assess the future unseen software changes. To replicate this scenario in the training and test phases, we split the data timewise. We set aside the data from the latest periods to use them for test and evaluation, and the rest of the data (early data) served as the training data.

More particularly, in the OpenStack dataset, we followed Hoang et al. (2019), Hoang et al. (2020), and Pornprasit and Tantithamthavorn (2021) and divided the entire dataset into five periods and used the data from the first four periods as the training set and the data from the last period as the test set. ApacheJIT has data from 17 years, from 2003 to 2019. In this dataset, we grouped the commits by year and, from the 17 years, kept the data of the first 14 years for the training set and the last three years for the test set. However, the data size for the previous three years was large (30,111 commits); therefore, we took a sample of 7,526 commits from the last three years of data and used it as the test set.

Similarly, the data of the first 14 years is large, and training the state-of-the-art model (JITLine) and JITGNN on it takes a lot of resources. To reduce the data size, we kept all the bug-inducing commits from the first 14 years and removed 31,729 clean commits to make a balanced training set. In this undersampling process, we did not randomly remove clean commits and removed them in a way that in every year of the training data, we have 50%. It is worth mentioning that although we used a balanced training set, we kept the test set as it was to make sure that it reflected a real-life scenario. Table 2 shows the statistics of the data used in our experiments.

5.2. Comparison models

In this part, we introduce the JIT models used in this study along with JITGNN.

5.2.1. Naive: Random classification

To better understand the results the following models achieve, we used a naive classifier. To this end, it is common to employ a random guessing model. Since the original OpenStack data in this study is not balanced, we used a biased random guessing model. General random guessing models assign labels to the samples in a uniformly random manner. In our biased random guessing model, the random label is biased towards the majority class in a way that represents the ratio of positive (bug-inducing commits) and negative (clean commits) samples in the training set. In other words, if the ratio of positive samples to all samples is r in the training set, the biased random guessing model

Table 2
Statistics of datasets.

Data	Set	Bug-inducing	Clean	Bug-inducing ratio	Total
OpenStack (Ours)	Training	1,325	7,839	0.1445	9,164
	Test	149	883	0.1443	1,032
ApacheJIT	Training	22,421	22,413	0.5000	44,834
	Test	1,448	6,078	0.1924	7,526

assigns 1 to each sample in the test set with a probability of r and assigns 0 with a probability of $1 - r$.

In the OpenStack v2 dataset, the ratio of bug-inducing to all commits is 0.1445 in the training set. Therefore, the naive classifier predicts samples in the test set as bug-inducing with a probability of 0.1445 and classifies them as clean with a probability of 0.8555. In the ApacheJIT dataset that we used, the training set was balanced; hence, the naive classifier labels samples in the test set in a uniformly random manner.

5.2.2. Baseline

Although the goal in McIntosh and Kamei (2018) is not to propose a JIT model, to conduct their experiments, they follow Zhou and Mockus (2011), Morales et al. (2015), Mcintosh et al. (2016) and employ multiple regression modeling with some techniques to capture the nonlinear relationship between the features. We used their publicly available model as our baseline model.

We studied McIntosh and Kamei (2018)'s scripts and made a few modifications to conduct our experiments on this model the same way we do on JITGNN. Originally, the multiple regression model in their work fitted in short-term and long-term settings due to the nature of the study. But in our study, we have a training set to fit the model on, and a test set to evaluate performance. Accordingly, we removed the periods in the regression model and fitted it on the training set, and tested it on the test set. Since the regression model only requires the commit metrics and commit metrics are available in OpenStack and ApacheJIT datasets, we did not need any further data processing. However, the ApacheJIT dataset does not contain the review information of commits while the features that are used in the multiple regression modeling in their work include the review metrics. Therefore, to successfully train the model, we removed the review features from it in the ApacheJIT experiments but used the same model as the original in the OpenStack experiments. For the rest of this section, we refer to the multiple regression model as the *Baseline*.

5.2.3. State-of-the-art

Pornprasit and Tantithamthavorn (2021) presented a JIT bug prediction approach called *JITLine*. In their paper, they compare the performance of JITLine with CC2Vec (Hoang et al., 2020), DeepJIT (Hoang et al., 2019), and EALR (Kamei et al., 2013). JITLine outperforms these models in most of the evaluation metrics. JITLine combines the commit metrics with the changed code (added and deleted code) in commits. More specifically, JITLine builds the feature vectors by joining the commit metrics features to the changed code's bag-of-words (BoW) representation. JITLine fits a random forest regression model on the feature vectors to predict the probability of commits being bug-inducing. They also use the SMOTE technique to oversample the bug-inducing commits and overcome the class imbalance problem.

As we mentioned above, JITLine uses the added and deleted code tokens in commits. The JITLine replication[4] package includes this data for OpenStack; therefore, we only needed to find the token data of the OpenStack commits we selected for our experiments (Table 2). However, ApacheJIT does not have the information on the added and deleted lines originally. Hence, we collected the added and deleted lines in the selected ApacheJIT commits and preprocessed them using the modules provided in the JITLine replication package.

[4] https://zenodo.org/record/4596503

While our experiments on the OpenStack dataset were completed successfully, due to the size of the ApacheJIT dataset and the BoW vocabulary set, both JITGNN and JITLine faced computing resource issues. Originally, both models use the entire vocabulary of code tokens to build the BoW vector representations. But in ApacheJIT experiments, to mitigate the memory consumption issue, we limited the JITGNN and JITLine vocabulary sets to 100,000 most frequent tokens. This number was chosen because both models could be trained with a reasonable amount of computing resources.

5.3. Evaluation metrics

To evaluate the discriminatory performance of JITGNN and the comparison models, we selected a combination of threshold-dependent and threshold-independent evaluation metrics that are commonly used in the literature (Kamei et al., 2013; Fukushima et al., 2014; McIntosh and Kamei, 2018; Hoang et al., 2019, 2020; Pornprasit and Tantithamthavorn, 2021).

AUC: AUC is the area under the receiver operating characteristic (ROC) curve. The ROC curve is the plot that illustrates the relationship between the true positive rate (TPR) and the false positive rate (FPR) over various thresholds. AUC is a threshold-independent metric that is used to assess the discriminatory power of binary classifiers. AUC is a value between 0 and 1. High AUC values represent higher TPR and lower FPR and demonstrate a robust discriminatory power of the classifier. A value of 0.50 AUC means that the classifier is randomly classifying the samples and values lower than 0.50 indicate that the classifier has negative classification power and the output of the classifier should be reversed (0 to 1 and 1 to 0).

Precision: Precision measures the power of a classifier to correctly detect positive samples. Precision is a threshold-dependent measure and in our study, we set the threshold to 0.50, meaning that if the output probability is 0.50 or higher, the model is classifying the sample as positive; otherwise, it is negative.

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

Recall: Recall measures the ability of a classifier not to miss positive samples. The recall is a threshold-dependent measure and similar to precision, we set its threshold to 0.50.

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

F_1 **score**: F_1 score is the harmonic mean of precision and recall. F_1 score is also threshold-dependent and we set the threshold to 0.50.

$$F_1 score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{6}$$

Matthew's correlation coefficient: Matthew's correlation coefficient (MCC) is a reliable metric that produces high scores if the binary classifier is performing well in all four categories in the confusion matrix. MCC ranges between −1 and +1. An MCC of +1 shows that the model's predictions are close to the actual label and −1 indicates a poor performance. We use this metrics for RQ2.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}} \tag{7}$$

Note that the threshold-dependent metrics (F_1 score, Precision, Recall, and MCC) we report in the next part are the metrics of the

positive class (bug-inducing commits) and the metrics of the negative class (clean commits) are not included.

The evaluation metrics above are the most widely used metrics for JIT models and can handle the class imbalance problem (Kamei et al., 2013; Yang et al., 2015; Fu and Menzies, 2017; McIntosh and Kamei, 2018; Hoang et al., 2019, 2020; Pornprasit and Tantithamthavorn, 2021). We used threshold-dependent metrics along with a threshold-independent one because ultimately, a potential tool implemented to alarm developers needs to decide whether a commit is bug-inducing or not. Although we set the default thresholds to 0.50, in RQ2, we also studied the evaluations for a range of different thresholds.

5.4. Hyperparameter tuning

Due to the complexity of the deep learning models and JITGNN in particular, it is usually not feasible to train the model with various possible values for every hyperparameter involved in its training. Regarding the training batch size, our available resources constrained us to use only batch sizes of 1. To identify optimal values for other hyperparameters, we used two approaches. Firstly, we trained the model with the values commonly used in similar works for certain hyperparameters, selecting the top-performing model based on its test set performance. These hyperparameters encompassed the model's number of training epochs (optimal: 10) and the size of its hidden layer (optimal: 32).

We adopted the frequently employed values for other hyperparameters that align with consensus in previous research. Specifically, we utilized the Adam optimizer with a learning rate of 0.001 and no weight decay. We adhered to the original formulations of activation functions and dropouts found in GCN (Kipf and Welling, 2017), the attention mechanism (Nair et al., 2020), and the neural tensor network (Socher et al., 2013), incorporating these components into our approach.

6. Results

6.1. RQ1: Can we replicate the baseline and the state-of-the-art JIT bug prediction models?

Approach: We studied the replication packages of the baseline and the state-of-the-art (JITLine). We ran the scripts on the original OpenStack dataset to compare the results with those reported in their respective papers (McIntosh and Kamei, 2018; Pornprasit and Tantithamthavorn, 2021).

To see how each component in JITLine affects the performance, we experimented with three variants of JITLine:

- *JITLine w/o Metrics* In this variant, the commit metric features are excluded from the JITLine original feature vectors fed to the random forest model.
- *JITLine w/o SMOTE*: This variant of JITLine does not have the SMOTE oversampling module.
- *JITLine w/o Metrics w/o SMOTE*: We discarded both the SMOTE oversampling module and the commit metrics in this variant. In other words, this variant solely works with the change codes (added lines and deleted lines) and builds a model by fitting a random forest regression model on the BoW representation of code tokens.

It should be noted that the SMOTE oversampling technique is part of JITLine and we did not use it in either building JITGNN or preparing the experiment datasets. To create a fully balanced dataset, we did undersample the clean commits (negative samples) in the ApacheJIT dataset, as we mentioned in Section 5.1.2.

Results: We were able to replicate the baseline model and JITLine results successfully. The results are in line with the ones reported in their respective papers. These results, along with the results of running the three variants on the original OpenStack dataset, are shown in Table 3.

Table 3
Replication result of the baseline model and JITLine on the original OpenStack dataset.

	AUC	F_1 score	Precision	Recall
Baseline	0.81	0.12	0.33	0.07
JITLine	0.83	0.37	0.47	0.30
JITLine w/o Metrics	0.79	0.41	0.39	0.42
JITLine w/o SMOTE	0.82	0.09	0.32	0.05
JITLine w/o SMOTE w/o Metrics	0.81	0.11	0.37	0.06

For the baseline, results showed an AUC of 0.81. Originally, McIntosh and Kamei (2018) reported an AUC of 0.72. The difference between our findings and those of McIntosh and Kamei (2018) can be attributed to the difference between the training and test sets. Our study employed JITLine's training and test sets, which are slightly different from the training and test sets in McIntosh and Kamei (2018) in this particular setting (long-term six-month setting). Therefore, we cannot compare our results with theirs. Additionally, our baseline evaluation encompassed precision, recall, and F_1 score metrics, which are not reported in the original paper.

Replicating JITLine leads to nearly identical outcomes to those detailed in the original paper. Running JITLine on our machines produces the same AUC value as reported in the original paper (0.83), along with marginally improved F_1 score, precision, and recall values. While the original scores were F_1 score=0.33, precision=0.43, and recall=0.26, we were able to achieve F_1 score=0.37, precision=0.47, and recall=0.30.

The results from the three variants of JITLine show that oversampling using SMOTE has a small impact on the discriminatory power of JITLine as the JITLine w/o SMOTE variant achieves an AUC of 0.82, only 0.01 AUC less than the full JITLine model. However, SMOTE affects the threshold-dependent metrics and F_1 score (0.09), precision (0.32), and recall (0.05) drop without oversampling. This observation means that oversampling impacts more on the threshold beyond which the model classifies a sample as bug-inducing or clean. The more balanced the training data is, the better the 0.50 threshold works as a distinguishing point.

6.2. RQ2: How does JITGNN perform compared to the baseline and the state-of-the-art JIT bug prediction models?

Approach: We conducted experiments involving a naive classifier, the baseline, the state-of-the-art JITLine, as well as the introduced JITLine variants from RQ1 and JITGNN on OpenStack v2 and ApacheJIT datasets. Similar to JITLine, we repeated the experiments for the following JITGNN architecture configurations:

- *JITGNN w/o Metrics*: We discarded the commit metrics from the ultimate change vector and fed the program change embedding derived from the neural tensor network (NTN) directly to the feed-forward neural network.
- *JITGNN Special Token*: In this variant, instead of using all the code tokens, we reduced the size of the vocabulary by using special tokens. Special tokens are defined for strings (<*STR*>), numbers (<*NUM*>), and arithmetic operations (<*OPE*>). More concretely, we replaced the raw strings, numbers, and operations with their corresponding special tokens. For example, if the types and values of two nodes in an AST are *String: "sample string one"* and *String: "sample string two"*, in this process, they both become *String: <STR>* to keep the vocabulary from getting too large.
- *JITGNN Supernode - Concatenation*: This variant architecturally differs from JITGNN. Originally, JITGNN uses an attention mechanism to obtain the global graph embedding followed by the NTN to output a single vector that captures the relationship between

Table 4
JIT bug prediction model results for bug-inducing commits (positive class).

	OpenStack (v2)					ApacheJIT				
	AUC	F_1 score	Precision	Recall	MCC	AUC	F_1 score	Precision	Recall	MCC
Naive	0.50	0.16	0.15	0.17	0.01	0.49	0.28	0.19	0.51	0.01
Baseline	0.78	0.14	0.35	0.08	0.07	0.78	**0.69**	**0.73**	0.65	0.36
JITLine	**0.79**	0.35	**0.45**	0.28	0.27	**0.81**	0.49	0.35	0.80	0.36
JITLine w/o Metrics	0.76	0.38	0.39	0.38	0.24	0.79	0.39	0.26	**0.82**	0.36
JITLine w/o SMOTE	**0.79**	0.11	0.41	0.06	0.20	**0.81**	0.50	0.36	0.81	0.37
JITLine w/o Metrics w/o SMOTE	0.78	0.10	0.39	0.06	0.22	0.79	0.38	0.25	0.81	0.36
JITGNN	**0.79**	0.35	0.44	0.28	0.26	**0.81**	0.50	0.37	0.78	0.38
JITGNN w/o Metrics	0.76	**0.43**	0.35	**0.54**	0.26	0.80	0.48	0.36	0.75	0.37
JITGNN Special Token	0.70	0.32	0.28	0.35	0.22	0.75	0.35	0.30	0.39	0.30
JITGNN Supernode - Concatenation	0.75	0.40	0.34	0.49	0.25	0.77	0.40	0.34	0.49	0.37

Table 5
JIT bug prediction model results for clean commits (negative class).

	OpenStack (v2)			ApacheJIT		
	F_1 score	Precision	Recall	F_1 score	Precision	Recall
Naive	0.86	0.86	0.86	0.63	0.81	0.52
Baseline	0.92	0.86	1.00	0.79	0.92	0.69
JITLine	0.91	0.88	0.95	0.77	0.93	0.65
JITLine w/o Metrics	0.90	0.90	0.90	0.76	0.92	0.65
JITLine w/o SMOTE	0.92	0.86	0.98	0.77	0.94	0.65
JITLine w/o SMOTE w/o Metrics	0.91	0.86	0.97	0.77	0.93	0.65
JITGNN	0.91	0.88	0.94	0.79	0.93	0.69
JITGNN w/o Metrics	0.87	0.91	0.83	0.72	0.78	0.68
JITGNN Special Token	0.84	0.88	0.79	0.67	0.76	0.60
JITGNN Supernode - Concatenation	0.86	0.89	0.83	0.70	0.75	0.66

the pre-change and post-change vectors. In this variant, however, the global graph embedding is the final hidden state of a dummy supernode that receives messages from all other nodes in the graph (similar to Scarselli et al. (2009), Li et al. (2017a)). Also, instead of NTN, the program change embedding is created by concatenating the global graph embedding of the pre-change and post-change subtrees.

Results: The results shown in Table 4 indicate that we cannot conclusively choose the best-performing JIT bug prediction model as the performance varies from one dataset and evaluation metric to another. As expected, the naive classifier achieves an AUC of almost 0.50 for both datasets, meaning it is randomly guessing the labels. In the OpenStack v2 dataset, JITGNN and JITLine achieve the same AUC score (0.79), which is 0.01 AUC better than the baseline. JITGNN and JITLine also achieve the same F_1 score (0.35) and recall (0.28), but the precision of JITLine is marginally higher (0.45 vs 0.44). Table 4 also shows the Matthew's correlation coefficient for the main models (i.e., Baseline, JITLine and JITGNN). The MCCs values of JITLine and JITGNN suggest a moderate level of correlation between the predicted and actual classifications. Nevertheless, these scores do not suggest a discernible advantage of any model over the other. The values in this table are based on the threshold of 0.50.

In ApacheJIT, JITGNN and JITLine share the first place in terms of AUC (0.81). However, the baseline performs better regarding the F_1 score (0.69) and precision (0.73). As for the recall, JITLine and its variants achieve the highest recall (0.80, 0.82, 0.81, 0.81 for JITLine, JITLine w/o Metrics, JITLine w/o SMOTE, and JITLine w/o Metrics w/o SMOTE respectively), outperforming the baseline (0.65) and marginally performing better than JITGNN (0.78). Although it is still lower than other models (not including the variants), the naive classifier achieves a surprisingly high recall (0.51) on the ApacheJIT dataset. The reason is that the training set is balanced, and the naive classifier labels samples in the test set as bug-inducing 50% of the time. Therefore, the naive classifier labels bug-inducing commits correctly almost 50% of the time.

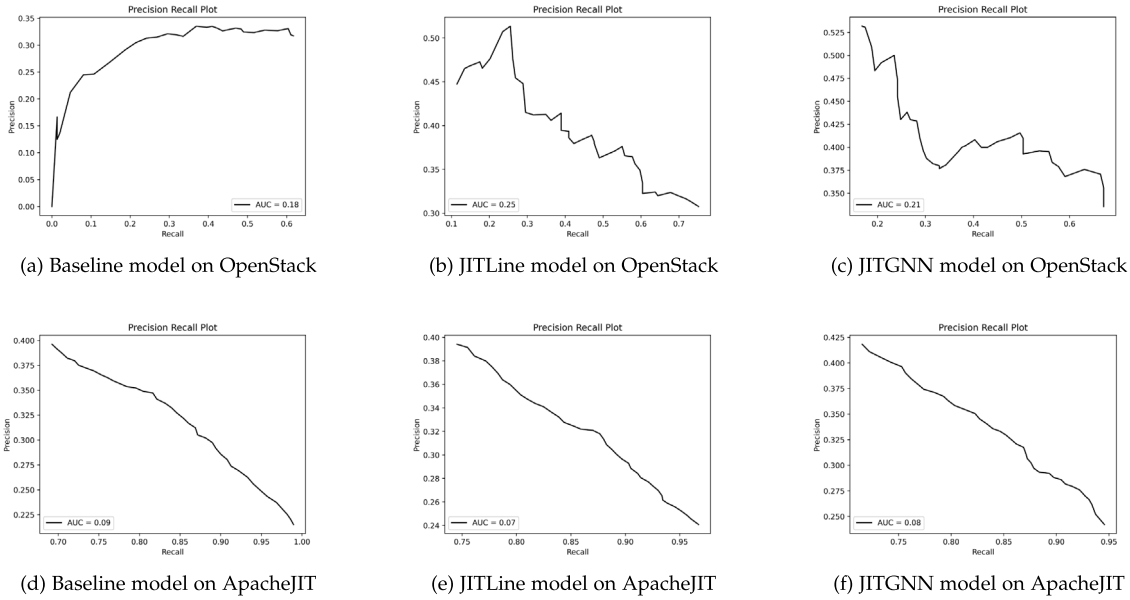
Fig. 3 shows the Precision–Recall plots for the three main models for both the OpenStack and ApacheJIT datasets. Notably, the Precision–Recall plot of the baseline model on the OpenStack dataset exhibits an unusual trend. This is attributed to the fact that the probability predictions of this model on this dataset do not exceed 0.55. For thresholds beyond this point, both precision and recall drop to 0. The corresponding AUC values of the Precision–Recall plots are also presented within the graph

The values presented in Table 5 display the metric results for clean commits, representing the negative class at the 0.50 threshold. The outcomes reveal that the models exhibited superior performance in the OpenStack v2 dataset compared to the ApacheJIT dataset for the negative class. In terms of a direct comparison, both JITLine and JITGNN exhibit similar scores.

Table 6 provides insight into how the three main models agree and disagree on the labels of the test samples by setting the threshold to 0.50. In this context, if the actual label of a sample is bug-inducing, the models that output a probability greater than or equal to 0.50 are "*Right*"; otherwise, they are "*Wrong*" in their predictions. On the other hand, the models are "*Right*" if they output a probability less than 0.50 for clean commits; otherwise, they are "*Wrong*".

Using these definitions, Table 6 categorizes the instances as follows: *ModelName Wrong* pertains to the cases where the model *ModelName* misclassified the test samples, while *ModelName Right* corresponds to situations where the model *ModelName* accurately classified the test samples. *ModelName* in this table is JITGNN, JITLine, or Baseline, as we compared the main three models for this part of the study and excluded the naive classifier. For example, the blue cell associated with **JITGNN Right, JITLine Wrong, Baseline Wrong** shows *572 (8%)*. It means that 572 of the ApacheJIT test set (8% of the set) are classified correctly by JITGNN, but JITLine and Baseline misclassified them.

Table 6 indicates that the predictions of the three models agree most of the time (*JITGNN Wrong, JITLine Wrong, Baseline Wrong* and *JITGNN Right, JITLine Right, Baseline Right* cells). In cases of disagreement, the numbers are close in the OpenStack dataset (22, 18, 28, 18, 26, 24). Nevertheless, disparities are evident within the ApacheJIT dataset.



(a) Baseline model on OpenStack (b) JITLine model on OpenStack (c) JITGNN model on OpenStack

(d) Baseline model on ApacheJIT (e) JITLine model on ApacheJIT (f) JITGNN model on ApacheJIT

Fig. 3. The Precision–Recall plots of the main models on the OpenStack and ApacheJIT datasets.

Table 6
The number of predictions the three models made right and wrong with respect to each other.

(a) OpenStack (Ours)

	JITLine wrong Baseline wrong	JITLine wrong Baseline right	JITLine right Baseline wrong	JITLine right Baseline right
JITGNN Wrong	92 (9%)	22 (2%)	18 (2%)	28 (3%)
JITGNN Right	18 (2%)	26 (2%)	24 (2%)	**804 (78%)**

(b) ApacheJIT

	JITLine wrong Baseline wrong	JITLine wrong Baseline right	JITLine right Baseline wrong	JITLine right Baseline right
JITGNN Wrong	1,344 (18%)	216 (3%)	91 (1%)	563 (7%)
JITGNN Right	572 (8%)	287 (4%)	251 (3%)	**4,191 (56%)**

JITGNN correctly predicts the labels of 572 test samples, while the other two models mispredict them. In the case of JITLine, it classifies 91 samples accurately, while the other two models misclassify them. The baseline only succeeds in labeling 216 samples correctly. In other words, JITGNN can accurately label more samples than the other two models.

Moreover, generally, JITGNN classifies the test samples correctly more than JITLine ($\langle 572+287+251+4, 191\rangle$ vs $\langle 563+91+251+4, 191\rangle$) and the baseline model ($\langle 572+287+251+4, 191\rangle$ vs $\langle 563+287+216+4, 191\rangle$) in this dataset.

We manually inspected the commits in the blue and red cells in Table 6. The blue cell corresponds to the commits JITGNN correctly classifies, but JITLine and Baseline classify them incorrectly. The red cell corresponds to the commits that JITGNN misclassified, but JITLine and Baseline classify correctly. We picked these two cells to see if we find a pattern in bug-inducing commits that existing models miss and the bug-inducing commits that JITGNN misses. Based on our observations, the only recurring pattern may be that JITGNN gives higher probabilities to small commits. In other words, we found more small commits among the commits in the blue cell that were bug-inducing, and only JITGNN classified them correctly. Among the commits in the red cell, most clean commits misclassified by JITGNN (JITGNN classified them as bug-inducing) were small.

To have a better understanding of the relationship between our threshold-dependent metrics (F_1 score, precision, and recall) and the threshold that these metrics use to identify bug-inducing commits, we repeated our experiments on the baseline, JITLine, and JITGNN for a range of thresholds between 0.20 and 0.70 with 0.05 increments. Table 7 shows the results of these experiments.

Predictably, the recall is high for low thresholds because many samples are classified as positive. Going toward higher thresholds, recall decreases, and precision increases generally. The exceptions are where the classifier gives low probabilities to true positive samples. In these cases, the samples whose probability of positiveness is beyond the threshold are not positive, which drops the precision. For example, in Table 7, the precision of the baseline drops at $threshold = 0.60$ and reaches 0 for the subsequent thresholds.

Overall, we can say that JITGNN performs marginally better than the baseline and is comparable to JITLine, the state-of-the-art. However, studying different thresholds indicates that software systems may need to choose different thresholds for their specific projects depending on how crucially bugs affect their systems and users.

6.3. RQ3: How does the size of the training set impact the performance of JITGNN on unseen data?

Approach: To answer this question, from the training set of the ApacheJIT dataset that we prepared for RQ2, we created subsets of training data with sizes varying from 10% to 100% of the original training set with 10% increments. Then, we trained JITGNN using these training sets, and for every training set, we evaluated the performance of the trained JITGNN on the original test set. Similar to RQ2, the ratio of bug-inducing commits in all these training sets was 0.50, while this ratio was 0.19 in the test set.

Table 7
Evaluations of threshold-dependent metrics for thresholds in [0.20, 0.70] with increments of 0.05.

(a) OpenStack (v2)

		0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70
Baseline	F_1 score	0.42	0.43	0.38	0.35	0.29	0.20	0.14	0.07	0.01	–	–
	Precision	0.32	0.36	0.34	0.36	0.37	0.36	0.35	0.31	0.16	0.0	0.0
	Recall	0.61	0.54	0.43	0.33	0.23	0.14	0.08	0.04	0.01	0.0	0.0
JITLine	F_1 score	0.43	0.43	0.45	0.43	0.43	0.38	0.35	0.29	0.21	0.16	0.10
	Precision	0.30	0.32	0.36	0.38	0.39	0.39	0.45	0.48	0.45	0.54	0.67
	Recall	0.77	0.66	0.61	0.51	0.46	0.36	0.28	0.21	0.13	0.09	0.05
JITGNN	F_1 score	0.46	0.46	0.43	0.37	0.36	0.35	0.35	0.28	0.24	0.20	0.16
	Precision	0.35	0.37	0.39	0.37	0.39	0.40	0.44	0.48	0.51	0.58	0.54
	Recall	0.68	0.58	0.48	0.38	0.34	0.30	0.28	0.20	0.15	0.12	0.09

(b) ApacheJIT

		0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	**0.65**	0.70
Baseline	F_1 score	0.68	0.71	0.72	0.72	0.72	0.71	0.69	0.67	0.64	0.61	0.57
	Precision	0.52	0.57	0.61	0.65	0.68	0.71	0.73	0.75	0.77	0.79	0.81
	Recall	0.98	0.94	0.88	0.83	0.77	0.71	0.65	0.60	0.55	0.49	0.44
JITLine	F_1 score	0.39	0.41	0.43	0.45	0.47	0.48	0.49	0.51	0.52	0.53	0.54
	Precision	0.24	0.26	0.28	0.30	0.32	0.33	0.35	0.37	0.40	0.43	0.47
	Recall	0.97	0.94	0.92	0.90	0.88	0.84	0.80	0.78	0.74	0.70	0.65
JITGNN	F_1 score	0.39	0.42	0.44	0.45	0.48	0.49	0.50	0.52	0.53	0.54	0.54
	Precision	0.24	0.27	0.29	0.31	0.33	0.35	0.37	0.40	0.43	0.45	0.48
	Recall	0.95	0.92	0.90	0.87	0.85	0.82	0.78	0.75	0.71	0.67	0.62

Table 8
JITGNN and JITLine performance on different sizes of ApacheJIT training set.

Size of ApacheJIT training data	JITGNN				JITLINE			
	AUC	F_1 score	Precision	Recall	AUC	F_1 score	Precision	Recall
4,483 (10%)	0.7768	0.4602	0.3366	0.7350	0.7860	0.4797	0.3455	0.7845
8,967 (20%)	0.7787	0.4617	0.3244	0.8004	0.7954	0.4891	0.3535	0.7935
13,450 (30%)	0.7767	0.4995	0.4449	0.5694	0.7947	0.4893	0.3560	0.7825
17,934 (40%)	0.7814	0.4945	0.4092	0.6246	0.7965	0.4614	0.3165	0.8508
22,417 (50%)	0.7988	0.4905	0.3550	0.7934	0.7974	0.4954	0.3593	0.7977
26,900 (60%)	0.7949	0.4984	0.3763	0.7383	0.7983	0.4676	0.3216	0.8564
31,384 (70%)	0.8018	0.4676	0.3260	0.8269	0.7987	0.4882	0.3513	0.7997
35,867 (80%)	0.8017	0.5048	0.3797	0.7530	0.8022	0.4922	0.3563	0.7956
40,351 (90%)	0.8042	0.5036	0.3794	0.7488	0.8007	0.4632	0.3174	0.8570
44,834 (100%)	0.8087	0.5038	0.3711	0.7844	0.8094	0.4960	0.3568	0.8135

Results: The results of these experiments are shown in Table 8. Based on these results, the discriminatory performance of JITGNN improves by increasing the size of the training set with few exceptions. However, this improvement is not sharp. As expected in deep neural networks, increasing the size of the training set helps JITGNN learn to adjust its parameters better to generalize on the unseen test set. However, the increase in the performance of JITGNN is not as sharp as expected, and the JITGNN model that was trained on our smallest training set (10% of the original training set) achieves an AUC of 0.78, which is marginally lower than the AUC of the JITGNN trained on the entire training set. Similar to AUC, the increases in other metrics are marginal.

To compare JITGNN with the state-of-the-art model (JITLine), we performed the same experiment with JITLine. Utilizing the same training samples across each ApacheJIT training subset and employing the identical test set, we trained JITLine. Table 8 indicates that increasing the size of the data the performance, generally, improves. The results show two difference between JITLine and JITGNN: (1) with smaller datasets as JITLine outperforms JITGNN, but as the data size grows, JITGNN exhibits more significant improvement. Ultimately, with the entire dataset, both models achieve almost the same performance. (2) generally by comparing the performance of model with the same data sizes, JITGNN demonstrates superior precision, while JITLine exhibits higher recall.

7. Discussion

In this work, we studied a JIT model that uses the graph structure of source codes by using ASTs of programs given to a GNN framework. We aimed to improve the discriminatory performance of JIT models by combining the code syntax, semantic information, and the change metadata. However, the findings presented in Section 5 indicate that JITGNN's performance is comparable with the current state-of-the-art JIT model (JITLine).

Although introducing code syntax and semantic information to the model does not seem to enhance the discriminatory capabilities of JIT models distinctly, our experiments yield the following interesting insights.

Oversampling influences the threshold-dependent metrics: The findings obtained in RQ1 from the three different versions of JITLine highlight an interesting pattern. While the application of SMOTE oversampling has a relatively minor influence on the overall discriminatory power of JITLine, there is a notable impact on threshold-dependent metrics. Specifically, when SMOTE oversampling is not used, the F_1 score decreases by 0.09, and precision and recall drop by 0.32 and 0.05, respectively (see Table 4). This trend implies that oversampling influences the threshold beyond which the model classifies a given sample. Thus, having a balanced training set results in higher values of threshold-dependent metrics at $threshold = 0.50$.

Also, the results in Table 7 show that the recalls at all thresholds are higher in the experiments on ApacheJIT. This observation is intuitive because the models see more positive (bug-inducing) samples in balanced training sets and are more inclined to output higher probabilities of defectiveness in general. This observation is important because systems that are more sensitive to bugs and fixing bugs is costly for them should consider high-recall JIT models.

Enhanced performance of JITGNN with small changes: Table 6 shows how the three main models we used in this study agree on their classification. The manual inspection of the commits correctly classified by JITGNN and misclassified by the other two models, and the commits correctly classified by the other two models and misclassified by JITGNN suggests that JITGNN tends to give higher probabilities of small changes. This information is important because it gives insight into the difference between the three models. Since JITGNN is a deep learning model at its core, the features that contribute to the final classification of the model cannot be examined to determine which feature attributes to misclassification. Therefore, this manual inspection is the only way to learn more about the classification pattern.

Performance of the cross-lingual JITGNN: The ApacheJIT dataset is a cross-project dataset, and JITGNN achieves an acceptable performance on this dataset (0.81 AUC, 0.50 F_1 score, 0.37 Precision, and 0.78 Recall). In another experiment, we evaluated the cross-lingual performance of JITGNN. In this experiment, we trained the *JITGNN Special Token* model on the ApacheJIT training data. Then, we collected the vocabulary of the training data in ApacheJIT, which is essentially Java AST types and values and manually mapped the ApacheJIT vocabulary to the Python vocabulary of the training data in OpenStack. Finally, we tested the JITGNN model trained on ApacheJIT on the test set of OpenStack v2.

We chose the *JITGNN special token* to reduce vocabulary size by converting all strings, numbers, and operations to the same type and value token and, consequently, the same embedding vector from BoW to set as node features. The ApacheJIT training set in this setting had 110 distinct tokens, and the size of the vocabulary of the OpenStack training set was 72. We mapped 49 tokens using the grammars of Java and Python. JITGNN performed poorly in this experiment (0.65 AUC, 0.13 F_1 score, 0.29 precision, and 0.08 recall).

Training bigger dataset size slightly boosts JITGNN performance: A larger training dataset generally allows a model to capture more diverse patterns and nuances in the data distribution. This aids in improving the model's ability to generalize well to new, unseen data. Although the results from RQ3 highlight an overall trend of enhanced JITGNN performance as the training set size increases, it does not display the higher performance that might have been anticipated initially. In JITGNN, as the training dataset size increases, the model's performance improvement does not scale linearly. This might be because, after a certain point, the model may start to memorize the data rather than understand the underlying relationships.

Interestingly enough, even when the training set is significantly reduced (10% of the original size), JITGNN still benefits from the essence of the data. This might be because even a subset of the data contains valuable patterns and trends, which the model can utilize for predictions. Hence, JITGNN's capability to make informed predictions with a smaller dataset could be particularly beneficial when collecting an extensive dataset is challenging or resource-intensive. Furthermore, in cases where real-time decision-making is crucial, like JIT models, having a model like JITGNN that can generate valuable predictions even with limited data might be valuable.

7.1. Implications and future research

The outcomes of this study can hold substantial implications for both JIT models and the broader landscape of software engineering research. These implications stem from exploring incorporating the source code structure into JIT models' prediction process and shed light on the potential ramifications for future research.

Refined model design: The study acknowledges that the effects of source code structure on JIT model predictions did not yield the expected improvements in JIT models. Sharing this outcome is invaluable for guiding fellow researchers away from potentially fruitless directions and toward more fruitful paths, ultimately saving time and resources in future endeavors. However, the observed marginal improvement of JITGNN over the baseline and its comparable performance to the state-of-the-art JIT model suggest that there might be a potential avenue for a better model design. Thus, exploring graph neural network architectures further could yield promising results. For example, JITGNN can potentially be improved by including more graphical program information. Data and control flow information can be added to the ASTs as new edge types. Adding various edge types has shown to be effective in software vulnerability detection (Zhou et al., 2019). Moreover, researchers could further investigate how to optimize the model's design to increase the speed of training, as one of the challenges we faced was the training time of JITGNN. We approached that challenge by reducing the size of the ASTs given to GCNs in JITGNN. Specifically, we removed the nodes unrelated to the changed nodes.

In-depth analysis of JITGNN predictions: An additional avenue that naturally emerges from this study is the opportunity to conduct an in-depth analysis of the classifications made by JITGNN. Our study shows that JITGNN's performance does not overcome state-of-the-art JIT models' performance, but researchers have the opportunity to delve into its interpretability. By scrutinizing the commits that JITGNN correctly and incorrectly classifies, researchers can study the specific types of bug-inducing commits easily identified by JITGNN and those it misses. This study would involve comparing how other JIT models perform in classifying the same commits to understand the strengths and limitations of JITGNN. Consequently, the findings could provide invaluable insights to guide further enhancements and refinements to JITGNN's bug detection capabilities.

Explainability of JITGNN predictions: While the explainability of the model was not a focus of this study, an interesting follow-up study could be the use of explainable artificial intelligence (XAI). The purpose of XAI is to enable humans to comprehend the logic behind an algorithm's output. XAI might offer pertinent explanations for JITGNN outcomes and empower practitioners to grasp the underlying insights. This could subsequently improve JITGNN's effectiveness and transparency. Previous works on JIT model predictions have used explainability methods such as LIME (Local Interpretable Model Agnostic Explanations) (Pornprasit et al., 2021; Pornprasit and Tantithamthavorn, 2021).

7.2. Threats to validity

7.2.1. Construct validity

Threats to construct validity are related to how proper our inferences are in this study based on the experiments and the results. Our results show that the GNN framework we designed and built for JIT bug prediction does not improve the discriminatory performance of existing JIT models that consider the changed code as plain text and do not include the graph structure of programs. This inference, however, is threatened by GNNs and ASTs as GNNs may not capture the structure of the code represented in ASTs well in the context of JIT bug prediction.

Recently, Rodriguez-Perez et al. (2020), Rodríguez-Pérez et al. (2020) introduced a group of bugs caused by external factors such as changes in requirements or APIs. They call this group of bugs *extrinsic bugs*. These bugs and their causes cannot be found using the SZZ algorithm (Śliwerski et al., 2005) that is employed to construct both datasets we used in this work. The authors suggest that researchers exclude this type of bug-inducing changes from the data they feed to JIT models. This approach improves the performance of JIT models, but on the other hand, JIT models cannot identify these bugs. Therefore, the results of this study cannot be applied in the context of extrinsic bugs.

7.2.2. External validity

Threats to external validity are about how the limitations of our experiments may threaten generalizability. In this study, we used two datasets: one is the OpenStack dataset presented in McIntosh and Kamei (2018) and used in its subsequent JIT bug prediction

studies (Hoang et al., 2019, 2020; Pornprasit and Tantithamthavorn, 2021), and the other dataset is a newly created JIT dataset called ApacheJIT (Keshavarz and Nagappan, 2022). ApacheJIT is a cross-project JIT dataset with historical change data from 14 popular Apache projects. ApacheJIT is a large dataset with more than 100,000 commits after an extensive filtering process suggested by da Costa et al. (2017) and McIntosh and Kamei (2018).

Although ApacheJIT is a large dataset containing commits from several different projects, there is still an external validity threat that we cannot necessarily generalize the results we obtained in our experiments to other software projects. Also, the primary language in the OpenStack dataset is Python, and in ApacheJIT is Java. One factor in the generalizability of our results is the programming languages in those projects. JITGNN works upon the AST of programs and the abstract information of program elements. But the same program in different languages may still have different ASTs depending on the language's grammar. Another limitation in our study is the computing resources for which we had to pick the 100,000 most frequent tokens in the vocabulary to be able to train both JITGNN and JITLine.

Finally, the network that we built uses GCNs as the GNN. Initially, we performed experiments with gated GNNs (GNNs) and changed them to GCNs due to their poor performances. GGNNs and GCNs are the most widely used GNNs in software engineering and programming languages, but other GNNs may work well for JIT bug prediction.

7.2.3. Internal validity

These threats relate to the uncontrolled factors that might have impacted our results. The datasets used in this work are constructed based on the SZZ algorithm (Śliwerski et al., 2005). SZZ is the most widely used algorithm to extract bug-inducing commits automatically in a software repository. However, it has its limitations (Rodríguez-Pérez et al., 2018; da Costa et al., 2017; Neto et al., 2018, 2019). To mitigate these limitations, da Costa et al. (2017) and McIntosh and Kamei (2018) propose filtering steps to remove suspicious bug-inducing changes. Both datasets used in this work were built by adhering to these filtering steps.

We trained JITGNN on a GPU machine. GPUs leverage parallel computing to increase the speed of training deep learning models. One downside of using a GPU is the inconsistency between different executions of programs. To mitigate this, we executed each experiment at least three times on our GPU and reported the result that at least two experiments agreed on. According to our observations, the scale of fluctuation in the final results was mainly less than 0.01 AUC, with some exceptions with a 0.01 difference (± 0.01 AUC).

Other internal validity threats in our study are the scripts used for our experiments. To build JITGNN, we mainly relied on verified scripts and modules for GCN, NTN, and the attention layer and modified certain parameters and specifications of these modules. Still, we wrote the scripts for the data preprocessing and connecting these modules. Moreover, for the replication study, we modified the scripts from McIntosh and Kamei (2018) to change them from a longitudinal model to a regular JIT model with one training set and one test set. Also, to experiment with different configurations of JITLine, we discarded the SMOTE and the commit metric connection in their scripts. With all these modifications, however, we believe our modifications did not erroneously change the nature of these models, as the results indicate similar or justifiably different performances from their original works.

Finally, we obtained the program ASTs from a tool called GumTreeDiff (Falleri et al., 2014). GumTreeDiff is a source code differencing tool that extracts the ASTs of programs written in the supporting languages and identifies the differences between the ASTs of two source codes.

8. Conclusion

In recent years, Just-In-Time (JIT) bug prediction models evolved from basic logistic regression models trained on historical commit metrics into models that include more information about the content of the change. Also, deep learning models are introduced to the bug prediction domain. Limited work has been done on deep learning in JIT bug prediction, but the proposed models have demonstrated huge potential. Following the recent interest in utilizing the graph structure of programs and applying graph neural network (GNN) models to them, we explore the effectiveness of GNNs in JIT bug prediction in this work.

We propose a deep GNN called JITGNN that learns vector representations of nodes in the abstract syntax trees (ASTs) of programs before and after the change to obtain vector representations of the program before and after the change using an attention mechanism. The pre-change and post-change vector representations are given to the neural tensor network (NTN) to combine them by generating a relationship vector. To benefit from the achievements of commit metrics in the past, the common 14 commit metrics widely used in the literature are collected and concatenated to the output vector of NTN. Finally, a feed-forward neural network takes the concatenated vector and outputs a probability that indicates how likely the commit will introduce bugs in the future.

We hypothesized that JITGNN will outperform the state-of-the-art in terms of the discriminatory power because it incorporates the source code of the changes while preserving the code structure. We compared the performance of JITGNN against multiple regression modeling (McIntosh and Kamei, 2018) as the baseline and JITLine (Pornprasit and Tantithamthavorn, 2021) as the state-of-the-art in four evaluation measures: Area under ROC curve (AUC), F_1 score, Precision, and Recall. The data we used in this study comes from two sources. One is the OpenStack data prepared by McIntosh and Kamei (2018), and the other is a newly built JIT dataset called ApacheJIT (Keshavarz and Nagappan, 2022). Our results show that JITGNN performance is comparable to the state-of-the-art, and our hypothesis is disapproved. We also investigated the impact of the size of the training set on the performance of JITGNN and learned that, predictably, increasing the size of the training set improves the performance. Still, the increases in evaluation metrics are not very sharp, and even with 10% of the ApacheJIT training set, JITGNN achieves acceptable performance.

CRediT authorship contribution statement

Hossein Keshavarz: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft. **Gema Rodríguez-Pérez:** Conceptualization, Methodology, Supervision, Writing – review & editing.

Declaration of competing interest

The nature of potential conflict of interest is described below:

- Meiyappan Nagappan and Shane Mcintosh from the University of Waterloo
- Gregorio Robles and Jesus M. Gonzalez Barahona from University Rey Juan Carlos
- Andy Zaidman from TUDelft
- Alexander Serebrenik from TUEindhoven

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Data availability

Data will be made available on request.

Acknowledgments

The authors would like to thank Prof. Meiyappan Nagappan, Prof. Shane McIntosh, Prof. Chakkrit (Kla) Tantithamthavorn, Prof. Yasutaka Kamei, Prof. Thong Hoang, and the Bank of Montreal (BMO).

The authors also acknowledge that our work takes place on the traditional territory of the Neutral, Anishinaabeg and Haudenosaunee peoples.

References

Allamanis, M., Barr, E.T., Bird, C., Sutton, C., 2015. Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, pp. 38–49. http://dx.doi.org/10.1145/2786805.2786849.

Allamanis, M., Brockschmidt, M., Khademi, M., 2017. Learning to represent programs with graphs, CoRR abs/1711.00740. Available: http://arxiv.org/abs/1711.00740.

Alon, U., Brody, S., Levy, O., Yahav, E., 2019a. code2seq: Generating sequences from structured representations of code. arXiv arXiv:1808.01400.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019b. Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3 (POPL), http://dx.doi.org/10.1145/3290353.

Arisholm, E., Briand, L.C., 2006. Predicting fault-prone components in a java legacy system. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. ISESE '06, Association for Computing Machinery, New York, NY, USA, pp. 8–17. http://dx.doi.org/10.1145/1159733.1159738.

Brockschmidt, M., 2020. GNN-FiLM: Graph neural networks with feature-wise linear modulation. In: III, H.D., Singh, A. (Eds.), Proceedings of the 37th International Conference on Machine Learning. In: Proceedings of Machine Learning Research, vol. 119, PMLR, pp. 1144–1152, Available: https://proceedings.mlr.press/v119/brockschmidt20a.html.

Bryan, J., Moriano, P., 2023. Graph-based machine learning improves just-in-time defect prediction. PLoS One 18 (4), e0284077.

Chen, D., Chen, X., Li, H., Xie, J., Mu, Y., 2019. DeepCPDP: Deep learning based cross-project defect prediction. IEEE Access 7, 184832–184848.

Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y., 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. 30 (3), http://dx.doi.org/10.1145/3436877.

Cho, K., van Merrienboer, B., Bahdanau, D., Bengio, Y., 2014. On the properties of neural machine translation: Encoder–decoder approaches. In: SSST@EMNLP.

da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E., 2017. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans. Softw. Eng. 43 (7), 641–657.

Dam, H.K., Pham, T., Ng, S.W., Tran, T., Grundy, J., Ghose, A., Kim, T., Kim, C., 2018. A deep tree-based model for software defect prediction, CoRR abs/1802.00921. Available: http://arxiv.org/abs/1802.00921.

El Emam, K., Melo, W., Machado, J.C., 2001. The prediction of faulty classes using object-oriented design metrics. J. Syst. Softw. 56 (1), 63–75.

Falleri, J., Morandat, F., Blanc, X., Martinez, M., Monperrus, M., 2014. Fine-grained and accurate source code differencing. In: ACM/IEEE International Conference on Automated Software Engineering. ASE '14, Vasteras, Sweden - September 15–19, 2014, pp. 313–324, Available: http://doi.acm.org/10.1145/2642937.2642982.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. Online, Association for Computational Linguistics, pp. 1536–1547, Available: https://aclanthology.org/2020.findings-emnlp.139.

Fu, W., Menzies, T., 2017. Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, pp. 72–83. http://dx.doi.org/10.1145/3106237.3106257.

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., Ubayashi, N., 2014. An empirical study of just-in-time defect prediction using cross-project models. In: Proceedings of the 11th Working Conference on Mining Software Repositories. In: MSR 2014, Association for Computing Machinery, New York, NY, USA, pp. 172–181, Available: https://doi-org.proxy.lib.uwaterloo.ca/10.1145/2597073.2597075.

Ghasedi Dizaji, K., Herandi, A., Deng, C., Cai, W., Huang, H., 2017. Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 5736–5745.

Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. In: Teh, Y.W., Titterington, M. (Eds.), Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. In: Proceedings of Machine Learning Research, vol. 9, PMLR, Chia Laguna Resort, Sardinia, Italy, pp. 249–256, Available: https://proceedings.mlr.press/v9/glorot10a.html.

Graves, T., Karr, A., Marron, J., Siy, H., 2000. Predicting fault incidence using software change history. IEEE Trans. Softw. Eng. 26 (7), 653–661.

Grover, A., Leskovec, J., 2016. Node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 855–864.

Guo, X., Liu, X., Zhu, E., Yin, J., 2017. Deep clustering with convolutional autoencoders. In: International Conference on Neural Information Processing. Springer, pp. 373–382.

Hassan, A.E., 2009. Predicting faults using the complexity of code changes. In: 2009 IEEE 31st International Conference on Software Engineering. IEEE, pp. 78–88.

Hassan, A.E., Holt, R.C., 2005. The top ten list: Dynamic fault prediction. In: 21st IEEE International Conference on Software Maintenance. ICSM'05, IEEE, pp. 263–272.

Hinton, G.E., Osindero, S., Teh, Y.-W., 2006. A fast learning algorithm for deep belief nets. Neural Comput. 18 (7), 1527–1554. http://dx.doi.org/10.1162/neco.2006.18.7.1527, arXiv:https://direct.mit.edu/neco/article-pdf/18/7/1527/816558/neco.2006.18.7.1527.pdf.

Hoang, T., Kang, H.J., Lo, D., Lawall, J., 2020. CC2vec: Distributed representations of code changes. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering. ICSE, pp. 518–529.

Hoang, T., Khanh Dam, H., Kamei, Y., Lo, D., Ubayashi, N., 2019. DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, pp. 34–45.

Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B., Hassan, A.E., 2010. Revisiting common bug prediction findings using effort-aware models. In: 2010 IEEE International Conference on Software Maintenance. pp. 1–10.

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2013. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. 39 (6), 757–773.

Keshavarz, H., Nagappan, M., 2022. ApacheJIT: A large dataset for just-in-time defect prediction. arXiv:2203.00101.

Khoshgoftaar, T.M., Allen, E.B., 2003. Ordering fault-prone software modules. Softw. Qual. J. 11 (1), 19–37.

Kim, S., Whitehead, E.J., Zhang, Y., 2008. Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. 34 (2), 181–196.

Kingma, D.P., Ba, J., 2015. Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. Available: http://arxiv.org/abs/1412.6980.

Kipf, T.N., Welling, M., 2017. Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations. ICLR.

LeClair, A., Haque, S., Wu, L., McMillan, C., 2020. Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension. Association for Computing Machinery, New York, NY, USA, pp. 184–195. http://dx.doi.org/10.1145/3387904.3389268.

Li, J., Cai, D., He, X., 2017a. Learning graph-level representation for drug discovery. arXiv preprint arXiv:1709.03741.

Li, J., He, P., Zhu, J., Lyu, M.R., 2017b. Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security. QRS, pp. 318–328.

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S., 2016. Gated graph sequence neural networks. In: Bengio, Y., LeCun, Y. (Eds.), 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2–4, 2016, Conference Track Proceedings. Available: http://arxiv.org/abs/1511.05493.

Ma, G., Ahmed, N.K., Willke, T.L., Yu, P.S., 2021. Deep graph similarity learning: A survey. Data Min. Knowl. Discov. 35 (3), 688–725.

Manjula, C., Florence, L.Z., 2018. Deep neural network based hybrid approach for software defect prediction using software metrics. Cluster Comput. 22, 9847–9863.

McIntosh, S., Kamei, Y., 2018. Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. IEEE Trans. Softw. Eng. 44 (5), 412–428.

Mcintosh, S., Kamei, Y., Adams, B., Hassan, A.E., 2016. An empirical study of the impact of modern code review practices on software quality. Empir. Softw. Eng. 21 (5), 2146–2189. http://dx.doi.org/10.1007/s10664-015-9381-9.

Mockus, A., Weiss, D.M., 2000. Predicting risk of software changes. Bell Labs Tech. J. 5 (2), 169–180.

Morales, R., McIntosh, S., Khomh, F., 2015. Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. SANER, pp. 171–180.

Moser, R., Pedrycz, W., Succi, G., 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. pp. 181–190.

Nadim, M., Mondal, D., Roy, C.K., 2022. Leveraging structural properties of source code graphs for just-in-time bug prediction. Autom. Softw. Eng. 29 (1), 27.

Nagappan, N., Ball, T., 2005. Use of relative code churn measures to predict system defect density. In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.. pp. 284–292.

Nair, V., Hinton, G.E., 2010. Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the 27th International Conference on International Conference on Machine Learning. ICML '10, Omni Press, Madison, WI, USA, pp. 807–814.

Nair, A., Roy, A., Meinke, K., 2020. FuncGNN: A graph neural network approach to program similarity. In: Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, ESEM '20, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3382494.3410675.

Neto, E.C., da Costa, D.A., Kulesza, U., 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 380–390.

Neto, E.C., Costa, D.A.d., Kulesza, U., 2019. Revisiting and improving SZZ implementations. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, pp. 1–12.

Nikolentzos, G., Meladianos, P., Vazirgiannis, M., 2017. Matching node embeddings for graph similarity. In: Thirty-First AAAI Conference on Artificial Intelligence.

Ostrand, T.J., Weyuker, E.J., Bell, R.M., 2004. Where the bugs are. ACM SIGSOFT Softw. Eng. Notes 29 (4), 86–96.

Ostrand, T.J., Weyuker, E.J., Bell, R.M., 2005. Predicting the location and number of faults in large software systems. IEEE Trans. Softw. Eng. 31 (4), 340–355.

Pan, K., Kim, S., Whitehead, Jr., E.J., 2006. Bug classification using program slicing metrics. In: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation. IEEE, pp. 31–42.

Pornprasit, C., Tantithamthavorn, C., 2021. JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, pp. 369–379.

Pornprasit, C., Tantithamthavorn, C., Jiarpakdee, J., Fu, M., Thongtanunam, P., 2021. Pyexplainer: Explaining the predictions of just-in-time defect models. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 407–418.

Qiao, L., Li, X., Umer, Q., Guo, P., 2020. Deep learning based software defect prediction. Neurocomputing 385, 100–110.

Qiao, L., Wang, Y., 2019. Effort-aware and just-in-time defect prediction with neural network. PLoS One 14 (2), e0211359.

Qin, H., Yan, J., Li, X., Hu, X., 2016. Joint training of cascaded CNN for face detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 3456–3465.

Rathore, S.S., Gupta, A., 2012. Investigating object-oriented design metrics to predict fault-proneness of software modules. In: 2012 CSI Sixth International Conference on Software Engineering. CONSEG, pp. 1–10.

Raychev, V., Vechev, M., Krause, A., 2015. Predicting program properties from "big code". SIGPLAN Not. 50 (1), 111–124. http://dx.doi.org/10.1145/2775051.2677009.

Rodriguez-Perez, G., Nagappan, M., Robles, G., 2020. Watch out for extrinsic bugs! A case study of their impact in just-in-time bug prediction models on the OpenStack project. IEEE Trans. Softw. Eng. 1.

Rodríguez-Pérez, G., Robles, G., González-Barahona, J.M., 2018. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. Inf. Softw. Technol. 99, 164–176.

Rodríguez-Pérez, G., Robles, G., Serebrenik, A., Zaidman, A., Germán, D.M., Gonzalez-Barahona, J.M., 2020. How bugs are born: a model to identify how bugs are introduced in software components. Empir. Softw. Eng. 25 (2), 1294–1340.

Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G., 2009. The graph neural network model. IEEE Trans. Neural Netw. 1 (20), 61–80.

Shihab, E., Hassan, A.E., Adams, B., Jiang, Z.M., 2012. An industrial study on the risk of software changes. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE '12, Association for Computing Machinery, New York, NY, USA, Available: https://doi-org.proxy.lib.uwaterloo.ca/10.1145/2393596.2393670.

Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories. MSR '05, Association for Computing Machinery, New York, NY, USA, pp. 1–5. http://dx.doi.org/10.1145/1083142.1083147.

Socher, R., Chen, D., Manning, C.D., Ng, A., 2013. Reasoning with neural tensor networks for knowledge base completion. In: Burges, C.J.C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K.Q. (Eds.), Advances in Neural Information Processing Systems, Vol. 26. Curran Associates, Inc.

Tan, M., Tan, L., Dara, S., Mayeux, C., 2015. Online defect prediction for imbalanced data. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. pp. 99–108.

Tompson, J.J., Jain, A., LeCun, Y., Bregler, C., 2014. Joint training of a convolutional network and a graphical model for human pose estimation. Adv. Neural Inf. Process. Syst. 27.

Viet Phan, A., Le Nguyen, M., Thu Bui, L., 2017. Convolutional neural networks over control flow graphs for software defect prediction. In: 2017 IEEE 29th International Conference on Tools with Artificial Intelligence. ICTAI, pp. 45–52.

Wang, S., Liu, T., Nam, J., Tan, L., 2020. Deep semantic feature learning for software defect prediction. IEEE Trans. Softw. Eng. 46 (12), 1267–1293.

Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: 2016 IEEE/ACM 38th International Conference on Software Engineering. ICSE, pp. 297–308.

Wills, P., Meyer, F.G., 2020. Metrics for graph comparison: a practitioner's guide. PLoS One 15 (2), e0228728.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S., 2019. A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. 32, 4–24.

Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J., 2015. Deep learning for just-in-time defect prediction. In: 2015 IEEE International Conference on Software Quality, Reliability and Security. pp. 17–26.

Yu, J., Zhao, K., Liu, J., Liu, X., Xu, Z., Wang, X., 2022. Exploiting gated graph neural network for detecting and explaining self-admitted technical debts. J. Syst. Softw. 187, 111219.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32.

Zhou, M., Mockus, A., 2011. Does the initial environment impact the future of developers. In: 2011 33rd International Conference on Software Engineering. ICSE, pp. 271–280.

Zimmermann, T., Nagappan, N., 2008. Predicting defects using network analysis on dependency graphs. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. pp. 531–540.