JITGNN: A Deep Graph Neural Network for Just-In-Time Bug Prediction

by

Hossein Keshavarz

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

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, while the entities that are predicted to be defective in other forms of bug predictions might be developed by multiple developers. JIT bug prediction can be applied when the design decisions are fresh in the developer's mind; therefore, it takes less effort to review the change and fix the potential issues. Over the years, many approaches have been proposed to tackle the JIT bug prediction problem. These methods mainly rely on the change metrics such as the size of the change, the number of modified files in the change, and the experience of the author.

Little work has been done on the inclusion of the syntax and semantics of the change in JIT models. Also, although there has been extensive work on employing deep learning models for other forms of bug prediction, there are not many deep models for JIT bug prediction. None of the existing JIT models in which the changed code is included, consider the graph structure of source codes and the change codes are considered as plain text in these models. In this work, we propose a JIT model that incorporates both the content and metadata of changes leveraging the graph structure of programs. To this end, 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 the state-of-the-art JIT models. Our study shows that JITGNN achieves the same AUC as the state-of-the-art model (JITLine), which does not consider the code structure of source codes, and they both have the same discriminatory power.

Acknowledgements

I would like to thank all the people who made this thesis possible. Especially my wonderful supervisor, Professor Meiyappan Nagappan, for his support and guidance throughout this process, and above all, being an amazing human being. I would also like to thank Professor Gema Rodríguez-Pérez, Professor Shane McIntosh, and Professor Yasutaka Kamei for offering me their time, support, and ideas. Finally, special thanks to Professor Shane McIntosh and Professor Yaoliang Yu for reviewing this thesis and offering me their helpful suggestions.

As a member of the University of Waterloo, I acknowledge that this work took place on the traditional territory of the Neutral, Anishinaabeg and Haudenosaunee peoples.

Dedication

I dedicate this work to my parents, my little sister, my older sister and her lovely children, and my friends, especially Sadegh.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software changes —whether they are fixing issues, adding features to the software, or improving existing features— are a major 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 issues as soon as possible, before users are exposed to them. One remedy for detecting such issues and fixing them before exposure is bug prediction. Bug prediction is studied in different levels of software structure (subsystems [53], modules [26], files [52]).

A popular trend in bug prediction is to predict bugs at the change-level [39]. This form of bug prediction is referred to as *Just-In-Time (JIT) Bug Prediction* [36]. The advantage of JIT bug prediction is that each software change is associated with only one developer and by predicting bugs immediately after a change is submitted, the developer still has the design decisions in his mind when alerts are raised, making it require less effort to find and fix the problems.

Over the years, many JIT bug prediction models (JIT models for short) were proposed. Commonly, they are machine learning models that are trained on historical data comprised of software changes with labels identifying whether a change introduced bugs into the system (bug-inducing) or not (clean). In the context of Git, we refer to software changes as *Commits*. These JIT models then are tested on unseen data to predict whether a commit is bug-inducing or not. This prediction can be either a binary classification or a regression problem where the goal is to predict the probability of a commit being bug-inducing.

The early JIT models were logistic regression models trained on a set of handcrafted features derived from commit metrics [35, 67, 47]. These metrics include information about different aspects of commits, such as the size of the change, the number of files, directories, and subsystems, the history of the changed files, and the experience of the commit author. The problem with using these metrics is that they do not include any information about the content of the change, and two arbitrary changes with similar metrics will be treated similarly without considering the change codes.

More recently, there have been works on the inclusion of the syntactic and semantic information of the changes into JIT models [30, 32, 58], and they achieved better performances. *JITLine* [58], for example, is the most recent work in this area, and it outperforms the prior models in most of the evaluation metrics the authors report. These approaches, however, consider the changed code as plain natural language text and do not consider the code structure of programs. Lately, using deep learning on the graph structure of programs derived from their abstract syntax trees (ASTs) has gained popularity in different areas of software engineering and programming languages [7, 8, 3, 54, 81, 13]. Some of these works leverage deep graph neural networks (GNNs) to directly apply novel GNN models, which are proved to be effective in other areas of computing, on program graphs.

In this work, we investigate applying a GNN on the JIT bug prediction problem to see if GNNs are effective in this domain. We built a deep GNN framework, called *JITGNN*, that performs JIT bug prediction by applying graph convolutional networks (GCNs) on the trees obtains from the AST of programs before and after the change. More specifically, JITGNN extracts the AST of the programs that are modified in a commit before and after the change and discards the AST nodes that are not changed or not directly connected to changed nodes to form change subtrees. Then, the pre-change subtree and the post-change subtree are fed as inputs to two 4-layer GCNs. The two GCNs learn vector representations of the nodes and two attention mechanism layers dynamically aggregate node vector representations to produce global graph-level vector representations. A neural tensor network (NTN) combines the graph-level representations of the pre-change and post-change AST subtrees and outputs a vector. This vector is concatenated with the commit metrics that conventionally have been used in JIT models. Finally, a feed-forward neural network outputs a probability that indicates how likely the commit is bug-inducing.

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

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

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

To answer the above research questions, we used the OpenStack dataset built by McIntosh and Kamei [47] and the ApacheJIT dataset that we collected and presented in Keshavarz and Nagappan [37] (more details in Chapter 3). We compared the performance of JITGNN with two existing JIT models. We used the multiple regression modeling that was adopted in McIntosh and Kamei [47] as the baseline model and JITLine [58] as the state-of-the-art JIT model. We evaluated the performance of the baseline, JITLine, and JITGNN by measuring the area under the ROC curve (AUC), F_1 score, precision, and recall.

After the performance analysis, we implemented an open-source JIT tool, called *Khata*, that developers can install on their machines and use to evaluate the riskiness of the staged changes in their Git working directory.

The contributions of this work are:

- Proposing the first deep GNN model for JIT bug prediction called JITGNN.

- Evaluating the performance of JITGNN against the baseline and the state-of-the-art JIT models.
- Investigating the impact of the size of the training set on the performance of our deep GNN model.
- Implementing a practical open-source JIT tool to evaluate software changes and report the riskiness probability of the changes to developers.

The organization of this thesis is as follows: In Chapter 2, we review the related work. Chapter 3 presents our work on building the ApacheJIT dataset. Chapter 4 introduces the datasets we used in this study. We explain the architecture and the specifications of JITGNN in Chapter 5. In Chapter 6, the experiment setups and results are presented. We discuss our results and introduce Khata in Chapter 7. And finally, we conclude this work in Chapter 8.

Chapter 2

Related Work

In this section, we review the literature on bug prediction in general and then give further overviews of the works that are done in Just-In-Time (JIT) bug prediction, deep learning models proposed for this problem, and using graph networks to tackle other software engineering problems involving the 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 [53], module-level [26], file-level [52], class-level [18, 9, 62], and function-level [83] are different levels of granularity for which researchers proposed methods to identify defect-prone entities before software issues are exposed. Kim et al. [38] performed combinatory research and used cached change location to predict bugs in different levels from directory-level to function-level based on the locality assumption. The features that are commonly used in these works are code complexity features but over time, change process features were proved to be better indicators of defect-prone entities and gained more popularity [27, 35].

In the context of bug prediction forms mentioned above, rather than a classification problem, bug prediction has been mainly 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). Another form of bug prediction is change-level, in which the problem is to classify the software changes that are defect-prone. The first work on change-level bug prediction goes back to Mockus and Weiss's work [50], where the authors 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. Kim et al. [39] followed Mockus and Weiss [50] and classified changes to clean and bug-inducing changes. They selected a wide variety of features to train an SVM classifier. These features include change message terms, the change terms from added and deleted lines, code complexity metrics, and change metadata (such as time, author, history).

2.2 JIT Bug Prediction

Kamei et al. [35] and Mende and Koschke [49] took the effort required to review fault-prone entities into consideration and worked on effort-aware bug prediction. Shihab et al. [67] conducted an industrial study on risky software changes. They made a distinction between bug-inducing changes and risky changes, which are not necessarily going to introduce bugs into the system but can have negative impacts on the software. They proposed an extensive set of change and code metrics and evaluated the importance of these factors.

Later, Kamei et al. [36] included the effort aspect in the change-level bug prediction and coined the term "Just-In-Time Quality Assurance" or "Just-In-Time Bug Prediction". In this work, they proposed a set of 14 factors in 5 dimensions, which were introduced by the previous work, that are the most important indicators of the defect-proneness of software changes. In this regard, they focused on change factors and discarded code factors. They trained an effort-aware logistic regression (EALR) on the data from 6 open-source and 5 commercial software projects.

Kamei et al. 's work [36] is a turning point in the direction of bug prediction research. Just-In-Time bug prediction attracted a lot of attention after this work and various aspects of this form of bug prediction were studied over the years. Fukushima et al. [23] attempted to solve the problem of small training data by training a cross-project model. They created a pool of training data from 11 different projects and selected the same 14 change metrics as Kamei et al. [36]; 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. [70] addressed two problems with the existing JIT bug prediction models by not adopting k-fold cross-validation (because it naturally does not comply with the real-world setting in JIT models) and applying 4 resampling techniques to overcome the class imbalance problem. McIntosh and Kamei [47] studied how the evolution of software and software changes impact the performance of JIT models in a longitudinal study. They built a carefully curated JIT dataset from the Git commits of two open-source projects and used multiple regression modeling to predict the probability of commits being bug-inducing.

Recently, a powerful JIT bug prediction model called *JITLine* was proposed [58]. JIT-Line compares its performance and training time against the existing state-of-the-art on the dataset that was built by McIntosh and Kamei [47]. JITLine beats the existing models in most of the evaluation metrics. Throughout this paper, we refer to JITLine as the state-of-the-art or JITLine interchangeably.

2.3 Deep Learning in JIT

Although there has been extensive work done on deep learning models in software defect prediction [73, 43, 72, 16, 46, 12, 74, 61], little research is conducted on deep learning in Just-In-Time bug prediction. The first work in this area is *Deeper* [77]. Adopting the same 14 change metrics as Kamei et al. [36], Yang et al. [77] used deep belief networks (DBN) [28] in this work to map the handcrafted features to feature representations in latent space. The latent feature representations, then, were given to a logistic regression model to perform the classification. The classification and feature extraction parts in Deeper are not end-to-end and they are not trained jointly.

Qiao and Wang [60] proposed a 3-layer neural network that trains upon 10 of the 14 change metrics to predict the probability of a given change inducing bugs in the future. They turned the JIT problem into a ranking problem and used 10-fold cross-validation; two settings that do not comply with the real-world JIT bug prediction. Another deep learning work on JIT bug prediction was done by Hoang et al. [30]. They proposed a model called *DeepJIT*, which unlike Deeper, is an end-to-end deep model and the deep networks for feature extraction and the deep networks for classification are trained jointly. DeepJIT is made up of two convolutional neural networks (CNNs) followed by max pooling layers for automated feature extraction and a feed-forward neural network to output a probability. The training data in DeepJIT is code changes and commit messages of the commits.

The authors of DeepJIT further explored the use of deep learning to learn vector representation of code changes in latent space and proposed *CC2Vec* [32]. JIT bug prediction is one of the three tasks the authors of CC2Vec used to evaluate the performance of their model. CC2Vec has a hierarchical attention network (HAN) to capture the hierarchical relationship between text entities in the commit log message and the code changes. HAN outputs one vector representation for the added lines and one for the deleted lines. These vectors are given to a comparison layer where a neural tensor network (NTN) [68] is applied on the two networks to output a single vector as the code change vector. 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. The problem with CC2Vec is that CC2Vec trains on the entire data including the test set, which is not an accepted practice because it does not reflect the real-world scenario where models are trained on the available data to predict the unseen observations in the future.

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 with deep learning. The gated graph neural network (GGNN) [45] is one of the earliest GNNs that was used in the realm of programs. In this work, the authors extended the basic GNN by adding GRU units [14] to update the hidden states of the nodes in the graph. One of the tasks they experiment with in this work is program verification by approximating reachable program states.

The use of GGNNs for program source codes was further studied by Allamanis et al. [3]. Predicting variable names and selecting the correct variables are two tasks that are approached in this work by employing GGNNs. This work attempted to represent programs with graphs derived from their abstract syntax trees (ASTs). They extended the ASTs by adding new edge types to capture control and data flow. 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 that deal with program source codes started to use program graphs and GNNs to take the graph structure of codes into consideration. These areas include program similarity [54], software vulnerability [81, 13], and code summary [42].

To the best of our knowledge, this 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 [83] employed network analysis on the dependency graph to find the defective binaries in Windows Server 2003. However, the context of our work is Just-In-Time bug prediction, we use ASTs, and our prediction is done by graph neural networks.

Chapter 3

ApacheJIT

JIT defect prediction models are machine learning models relying on historical data. They require a set of past change revisions with each revision being identified whether or not it introduced a bug to the software (bug-inducing). In addition to change revisions and change labels, JIT defect prediction datasets often come with change metrics that have proved to be helpful in analysis and prediction [36, 78, 47].

Over the past few years, deep learning models found their way to JIT defect prediction [30, 32]. Although deep learning models have demonstrated solid performances in other areas of computing [59, 4, 17], DeepJIT [30] and and CC2Vec [32] do not outperform simple methods like logistic regression [47, 65]. This can be attributed to two main reasons. First, there are not many JIT datasets publicly available and most of the existing ones are small; while deep learning models are more effective when the size of the training data is large [82, 10, 84]. Secondly, the number of bug-inducing changes in the lifetime of a software system is often smaller than the number of clean changes. This leads to the class imbalance problem in JIT datasets. Undersampling the majority class makes the dataset even smaller and oversampling the bug-inducing class introduces bias. Deep learning models are more sensitive to both [34].

For example, one of the most widely used datasets for JIT defect prediction is presented by McIntosh and Kamei [47]. Although this dataset consists of carefully curated change revisions in *QT*[1] and *OpenStack*[2] projects, it has 25,150 QT changes and 12,374 OpenStack changes, and the ratios of bug-inducing changes to total changes are 8% and 13% for QT and OpenStack respectively.

We built ApacheJIT, a large dataset for JIT defect prediction. ApacheJIT consists of software changes in popular Apache projects. These changes have been selected carefully after applying filtering steps recommended in the literature [15, 47]. ApacheJIT has 106,674 commits (28,239 bug-inducing, 78,435 clean). ApacheJIT is one of the largest available JIT defect prediction datasets and it is suitable for JIT models that require a large number of software changes with many bug-inducing changes.

[1]https://www.qt.io/

[2]https://www.openstack.org/

3.1 Previous JIT Datasets

In this section, we review the previous Just-In-Time (JIT) defect prediction datasets. We found four major JIT datasets in the literature that are large or used in multiple studies.

Kamei et al. [36] performed a large-scale study of change-level defect prediction and coined the term *Just-In-Time Quality Assurance*, which evolved into *Just-In-Time Defect Prediction*. They investigated the effectiveness of logistic regression on detecting bug-inducing changes in 6 open-source and 5 commercial software projects. They extracted the changes from CVS and linked the fixing changes to the issues in the issue tracking systems. They used the basic SZZ algorithm [85] to label bug-inducing changes (except for two open-source projects that did not have issue keys in their change comments and they used Approximate SZZ). The dataset is not publicly available.

Jiang et al. [33] attempted to separate the prediction for different developers and called this problem *Personalized Defect Prediction*. They built a dataset of Java and C/C++ source codes from 6 open-source projects. The bug-fixing changes of two projects were previously manually found and for the rest of the projects, they applied keyword search. They labeled bug-inducing changes using SZZ without applying any filtering. Although their dataset has been used in Tan et al. [71] and Wang et al. [75], these works are done by the same team and the data is not publicly available.

McIntosh and Kamei [47] conducted a longitudinal study on JIT bug prediction models to see how the performance of JIT models changes over time. They built a dataset of 37,524 commits from OpenStack (12,374 commits) and QT (25,150 commits). They extended their work in [15] and applied a set of filtering steps on SZZ to remove the false positive bug-inducing changes. The replication package of the study and the datasets are available. This dataset has been widely used to evaluate JIT models.

Fan et al. [20] investigated the impact of mislabeled changes labeled by four SZZ variants (Basic SZZ, AG-SZZ, MA-SZZ, RA-SZZ). They claim that RA-SZZ generates the cleanest labels and used this variant as the baseline. They did not include McIntosh and Kamei's variant [47] in the study because it does not address false negatives (due to code indentation). They built a dataset of 10 Apache projects with 126,526 commits. RA-SZZ (the baseline) identifies 13,078 bug-inducing commits in this data. Although the authors have made the data available, the link between revision IDs and bug-inducing labels is missing. To the best of our knowledge, this data is not used in any JIT model evaluation.

In this work, we adopt the McIntosh and Kamei's approach [47] to identify bug-inducing commits because the dataset has been widely used to evaluate JIT models [30, 32, 58, 24].

3.2 ApacheJIT Dataset & Usage

ApacheJIT is one of the largest available datasets for JIT defect prediction. This dataset is a collection of carefully selected and filtered software changes in a set of popular Apache projects. ApacheJIT includes 106,674 software revisions from 2003 to 2019. These change revisions are derived from the issue reports from January 1, 2010, to December 31, 2019. 28,239 of these revisions are labeled as bug-inducing through the process explained in Section 3.3. ApacheJIT is suitable for defect prediction models that require a large set of historical data to learn prediction models.

In particular, ApacheJIT can be used to train deep learning models that require large datasets for effectively capturing the patterns in the historical data and using them to accurately predict future observations. Currently, the performance of deep learning models on JIT defect prediction datasets is not as promising as their performance in other areas of computing. One reason is that available JIT defect prediction datasets do not contain many samples and consequently, the number of bug-inducing changes models see during the training is small.

In addition to identifying whether or not each change revision has introduced bugs into systems, the data presented in this work also includes some of the change metrics that are commonly used for JIT defect prediction. The following is the list of these metrics (columns of the datasets):

change date, # of lines added, # lines deleted, # files touched, # directories touched, # of subsystems touched, change entropy, # of distinct developers touched files, the average time from last change, # of unique changes in files, change author experience, change author recent experience, change author subsystem experience.

The explanation of each metric is presented in Table 3.1. We used the same approach as Kamei et al. [36], who proposed this set of metrics, to obtain the change metrics in this work. Table 3.2 shows the statistics of ApacheJIT. The ApacheJIT dataset and the related scripts are publicly available[3].

3.3 Data Construction

The major part of constructing the ApacheJIT dataset is finding bug-inducing commits. This part was done based on the SZZ algorithm [85]. The SZZ algorithm has been widely used to detect bug-inducing commits, and in this work, we used it with some modifications. The SZZ algorithm starts with collecting the issue reports that are marked as fixed. Then these fixed issue reports are linked to their corresponding fixing commits. Finally, from the lines changed in the fixing commits, potential bug-inducing commits are detected.

Initially, we selected 15 popular Apache projects that had many bug reports (we used the data of 14 projects in the end). Our measure of popularity in this selection was the number of stars each project has on GitHub. Table 3.3 shows the selected projects.

[3]https://doi.org/10.5281/zenodo.5907001

Table 3.1: 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

Table 3.2: The number of collected issues and the statistics of ApacheJIT. Percentages under the Bug-inducing column indicate the ratio of bug-inducing commits to total commits.

Project	**Issues**	**Bug-inducing**	**Clean**	**Total**
ActiveMQ	1,967	1,404 (23%)	4,722	6,126
Camel	3,276	3,078 (14%)	19,622	22,700
Cassandra	5,358	3,117 (38%)	5,042	8,159
Flink	4,166	2,811 (24%)	8,880	11,691
Groovy	2,549	1,614 (20%)	6,445	8,059
HDFS	3,672	2,222 (21%)	8,137	10,359
HBase	7,085	3,782 (43%)	4,948	8,730
Hive	7,931	4,223 (61%)	2,619	6,842
Ignite	3,256	2,439 (20%)	9,597	12,036
MapReduce	2,080	838 (15%)	4,995	5,833
Mesos	2,955	-	-	-
Kafka	3,038	1,115 (46%)	1,269	2,384
Spark	7,648	632 (43%)	833	1,465
Zeppelin	1,089	622 (42%)	829	1,451
Zookeeper	859	342 (40%)	497	839
Total	**56,929**	**28,239 (26%)**	**78,435**	**106,674**

Table 3.3: Selected Apache projects in this study

ActiveMQ	Camel	Cassandra	Flink	Spark
Zeppelin	Groovy	Hadoop HDFS	HBase	Hive
Zookeeper	Ignite	Hadoop MapReduce	Mesos*	Kafka

* removed after fixing commit collection step (Section 3.3.2).

3.3.1 Bug Report Collection

As explained above, SZZ starts with collecting issue reports. All the Apache projects we selected keep their issue reports on Apache's JIRA Issue Tracker[4]. On JIRA, after selecting the aforementioned projects, we applied further filtering. First, we narrowed down our study focus to the issue reported from January 1, 2010, to December 31, 2019. Next, we filtered out the issues that were not identified as *bug*s. And finally, we picked the issues that are now marked as fixed. On JIRA, these issues are the issues with Status set to *Closed* or *Resolved* and with Resolution set to *Fixed*. Finally, after applying the filtering steps mentioned above, we had 56,929 bug reports. Table 3.2 shows the number of bug reports (issues) in each project.

The choice of the time period and the number of projects was based on the initial goal we set before constructing the dataset to finally collect almost 100,000 commits. A dataset of this size is ideal because it contains enough samples to extract patterns from and has many bug-inducing commits. Plus, learning a machine learning model on such a dataset is computationally feasible but many models face computation challenges when they are trained on larger datasets.

We should note that increasing the time period or the number of projects would result in a decrease in the other factor. We also took the bias-variance trade-off into account for project and time selection. In theory, if we have commits from more projects over a short period of time, the samples will have high variance and this helps the generalizability of the model after the training. However, such a diverse dataset may result in a model with a high bias error and the models might not be able to fit well on the data.

3.3.2 Fixing Commit Collection

After obtaining the issue reports that have been fixed, we looked for the commits that fixed these issue reports in the version control system (VCS). We followed the approach in Kamei et al. [36] and McIntosh and Kamei [47]. Each issue is identified uniquely with an issue key on JIRA. With the help of this identifier, for each project, we searched through all the commits on the main branch of the project Git repository and looked for commits whose commit messages indicate the change is fixing one of the issue keys we collected. This approach works because conventionally, developers add the issue key of the bugs they fix to the commit message.

In previous work, the search for these commits is done manually by looking for keywords in the result of the *git log* command [85, 36, 47]. In this work, however, we utilized GitHub search. This was feasible because all the projects we selected are stored on GitHub. The reason we preferred GitHub search to manual pattern matching was that the GitHub search engine returns the best match if it exists. This is especially useful when a commit message of a fixing commit does not include the issue key in the expected format. We compared the result of the GitHub search with the result of string matching on *git log* outputs and found

[4]https://issues.apache.org/jira/

out that overall, the commits returned by GitHub search are more relevant. To search on GitHub, we used the GitHub REST API[5].

In this process, if there is no commit with a commit message in which an issue key is, GitHub returns no commit. If there are commits with commit messages containing an issue key, GitHub returns one or several commits. We could identify the following reasons for the latter case:

1. Developers make several attempts to fix an issue before closing the bug report because the first attempts are *not enough*.
2. Developers make several attempts to fix an issue before closing the bug report because the first attempts are *incorrect*.

The scenarios mentioned above make finding true fixing commits challenging in both pattern matching and GitHub search approaches. On one hand, one may decide to include all the commits returned for one issue key because they are all related. On the other hand, one may only consider the most relevant of the multiple commits as the fixing commit. In this work, we chose the latter direction and picked the latest commit as the fixing commit. Our justification was that by picking all the aforementioned commits, later SZZ will consider many clean commits as bug-inducing (case 2 above). Therefore, the ultimate dataset will have high false-positive bug-inducing commits.

Among all the commits returned for one bug report, we found the latest one to be the most relevant. By picking the latest commit as the fix commit, we are almost certain that the commit we have picked is truly a fixing commit and can be used to find bug-inducing commits based on SZZ. This approach, however, leads to missing some bug-inducing commits (case 1 above), and consequently, our ultimate dataset will have higher false-negative commits (bug-inducing commits that are labeled as clean). Essentially, this is a trade-off between more false positives and more false negatives, and in this study, we chose the latter.

After collecting fixing commits as described above for all 15 projects we noticed that the commit messages in Apache Mesos do not comply with the conventional format and even GitHub search was not able to find fixing commits. Therefore, we eliminated all Apache Mesos data and continued with the remaining 14 projects.

3.3.3 Finding Bug-inducing Commits

At the end of the previous step, we linked 44,202 commits in the 14 projects to issue keys. These commits represent the fixing commits for the collected issues. The next step is to use these fixing commits to find bug-inducing commits.

[5]https://docs.github.com/en/rest

Git Annotate

In this step, each fixing commit is traced using *git annotate* command. This command annotates all lines of a given file showing the last revision that touched the line. For each fixing commit, we run *git annotate* on the files modified in the commit and get the last revisions that touched the deleted lines before the fixing commit. To implement the described process, we used the SZZ tool in the PyDriller framework[6] [69]. The SZZ tool in PyDriller gets a commit and returns the commits that last touched the deleted lines of the files modified in the given commit.

Filtering

The basic version of SZZ has limitations. The SZZ algorithm tends to label many clean commits as bug-inducing. Accordingly, we performed some heuristics to reduce false positives. We followed the filtering discussed in da Costa et al. [15] and McIntosh and Kamei [47] to filter out linked bug-inducing commits that are likely to be clean.

1. We made fixing commit - bug-inducing commit pairs and associated each with the issue key corresponding to the fixing commit. We removed the pairs where the bug-inducing commit date was after the issue report date (the date the issue was created on JIRA). Note that we did not remove the bug-inducing commit or the fixing commit separately as they may show up in other pairs and end up as valid bug-inducing and valid fixing commits respectively. This step filtered 5,048 bug-inducing commit candidates.

2. At the end of the *git annotate* process each fixing commit may be linked to several bug-inducing commits (a fixing commit may fix several bugs). We call the number of bug-inducing commits each fixing commit is linked to *fixcount*. da Costa et al. [15] and McIntosh and Kamei [47] filter out fixing commits whose fixcounts are more than a threshold. They refer to these commits as *suspicious fixing commits*. In their works, the threshold is set to upper Median Absolute Deviation (MAD) of fixcounts.

   $$upperMAD = M + median(|M - X_i|), \tag{3.1}$$

   where M is the median of fixcounts and X_i is the fixcount of commit i.

   In the present work, however, the upper MAD was too small, and choosing it as the threshold would filter too many fixing commits. As an alternative, we chose the sum of the mean and standard deviation of fixcounts as our threshold. Note that again, this is a trade-off between high false-positive and high false negative. Filtering out too many fixing commits will cause more bug-inducing commits to be labeled as clean commits in later steps. This step filtered 12,165 commits.

3. Similarly, we can define *bugcount* as the number of fixing commits each bug-inducing commit is linked to. This means that one commit has introduced multiple bugs into

[6] https://pydriller.readthedocs.io/

the system (multiple bug reports) and each has been fixed by a fixing commit. Again, to filter out the suspicious bug-inducing commits, we set a threshold of $mean + std$ of bugcounts. This step removed 1,257 bug-inducing commit candidates.

4. Following McIntosh and Kamei [47], we removed large commits. By large commits, here, we mean the commits that modify more than 100 files or have more than 10,000 lines of changed code. Lines of changed code is the total number of lines that were removed or added through the commit. This step removed 890 bug-inducing commit candidates.

5. In this work, we focused on Java programming language and built a uniform Java dataset. This language constraint makes it feasible to do static analysis on the source codes (our static analysis is explained in the next part). Among the selected projects, Java is the language in which most of the repository files are written. Therefore, we picked Java and filtered out the commits that do not modify any Java source code. 10,251 commits were filtered.

6. In order to avoid trivial changes, we performed a static analysis on the abstract syntax trees (ASTs) of the changed source code. In this process, we compared the AST of each Java program that was changed in a commit before and after the change. If there was at least one node in either of the two ASTs without a match in the other AST, we mark the change as non-trivial and keep the Java program; otherwise, the change is trivial. If all the Java programs in a commit have been changed trivially, we remove the commit. To conduct this analysis, we used GumTreeDiff [19]. GumTreeDiff is a tool that finds the differences between two source codes written in the same language using their ASTs. Examples of trivial changes are changes that modify comments, white spaces, and string or numeric literals. 274 commits were removed in this step.

Before the applying filtering steps, there were 58,124 bug-inducing commit candidates. 29,885 commits were filtered and the 28,239 commits remaining were labeled as bug-inducing commits. Table 3.2 shows the number of bug-inducing commits in each project. It is worth mentioning that the number of bug-inducing commits in Apache Spark is low with respect to the number of issue reports in this project. The reason is that Java is not the major programming language in Apache Spark.

3.3.4 Finding Clean Commits

In addition to the commits that introduce bugs into the system, we also would like to know what changes are safe. We call these changes *clean* commits. In projects, usually, the clean commits outnumber bug-inducing commits because most changes are reviewed by project developers other than the author before they are integrated into the system.

We collected all the commits in the selected project repositories from the date of the earliest bug-inducing commit (Sep 11, 2003) to the date of the latest one (Dec 26, 2019). We removed the commits we already had labeled as bug-inducing and the fix commits that had at least one corresponding bug-inducing commit. The number of remaining commits was 149,962. We also applied filtering steps (4) and (5) in Section 3.3.3 to make the filtering process similar to bug-inducing commits (steps (1)-(3) are not applicable). Finally, we had 78,435 clean commits.

3.3.5 Commit Metrics

In the end, we also added the set of common change metrics defect prediction datasets have. The metrics we collected were the same as the ones discussed in Kamei et al. [36]. We followed the same steps.

Chapter 4

Dataset

In this section, we introduce the Just-In-Time (JIT) bug prediction datasets that are used in this study and give some details about them.

4.1 OpenStack

JIT bug prediction models proposed in recent years [30, 32, 58] evaluated their performances on the dataset that was presented in McIntosh and Kamei [47]. The authors, in this work, study the impact of time on the discriminatory performance and calibration of JIT bug prediction models. They extracted 37,524 change revisions from QT and OpenStack projects for their study and made the dataset publicly available. This dataset has 25,150 QT changes mainly written in C/C++ and 12,374 OpenStack changes written in Python.

The commit ID of the change revisions are used as unique identifiers in this dataset and each revision has a label that identifies whether the change introduced bugs into the system or not (is bug-inducing). The ratio of bug-inducing changes to total changes is 8% and 13% for QT and OpenStack respectively. In addition to 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 is a combination of 14 change metrics used by Kamei et al. [36] and new change metrics related to the change review process.

In the present work, we used the OpenStack subset of this dataset to replicate the baseline and the state-of-the-art JIT bug prediction models and to evaluate and compare the performance of JITGNN with existing JIT models (RQ1 and RQ2). For RQ3 (the impact of the size of the training set on JITGNN), however, we did not use the OpenStack data because it is not large enough to perform experiments with a wide range of different sizes.

4.2 ApacheJIT

A limitation of McIntosh and Kamei's dataset [47] and other available JIT bug prediction datasets is that they have a small number of bug-inducing commits. The reason is that naturally, in a software project, changes are reviewed and they are accepted if the reviewers agree and the tests pass successfully. The process of accepting a change makes the number of defective accepted changes small. However, even with this process in place, there are still defective changes that are accepted, and JIT bug prediction attempts to identify these changes. JIT bug prediction models are mainly machine learning models that are 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 bug prediction model, the model should see many positive samples, which in our context are bug-inducing samples, to be able 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. To overcome this problem, we built a large cross-project JIT bug prediction dataset called *ApacheJIT* from the commits of 14 popular Apache projects. ApacheJIT has 106,674 commits, 28,239 of which are labeled as bug-inducing. Commits in ApacheJIT have the same set of attributes as the features in Kamei et al. [36]. ApacheJIT is presented in Keshavarz and Nagappan [37] and Chapter 3. In this study, we used ApacheJIT for all three research questions.

Chapter 5

JITGNN Framework

In this chapter, we present the *JITGNN* framework. JITGNN is a deep graph neural network (GNN) model for Just-In-Time (JIT) bug prediction. Like other JIT models, given a commit, JITGNN assesses the changes that were made in the commit to inform the commit author how likely the commit will introduce bugs into the software. JITGNN makes this prediction using both the content of the change (changed code) and the change metadata (commit metrics). The implementation of JITGNN is publicly available[1].

5.1 Overview

To include the syntactic and semantic information of the changed code in its assessment, JITGNN uses graphs to model programs that are changed in the given commit. JITGNN exploits abstract syntax trees (ASTs) to represent programs with graphs. To assess a program change, JITGNN builds one graph from the AST of the program before the change and one graph from the AST of the program 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 program before the change and the change subtree of the program after the change are fed into two graph neural networks (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, next, are fed into a neural tensor network (NTN) unit, which is a bilinear neural network, to compute a vector of the relationship between the two embeddings. A vector of commit metrics 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. Figure 5.1 demonstrates the architecture of JITGNN. JITGNN is an end-to-end model that is 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

[1] JITGNN implementation is available at: https://github.com/hosseinkshvrz/jit-bugpred

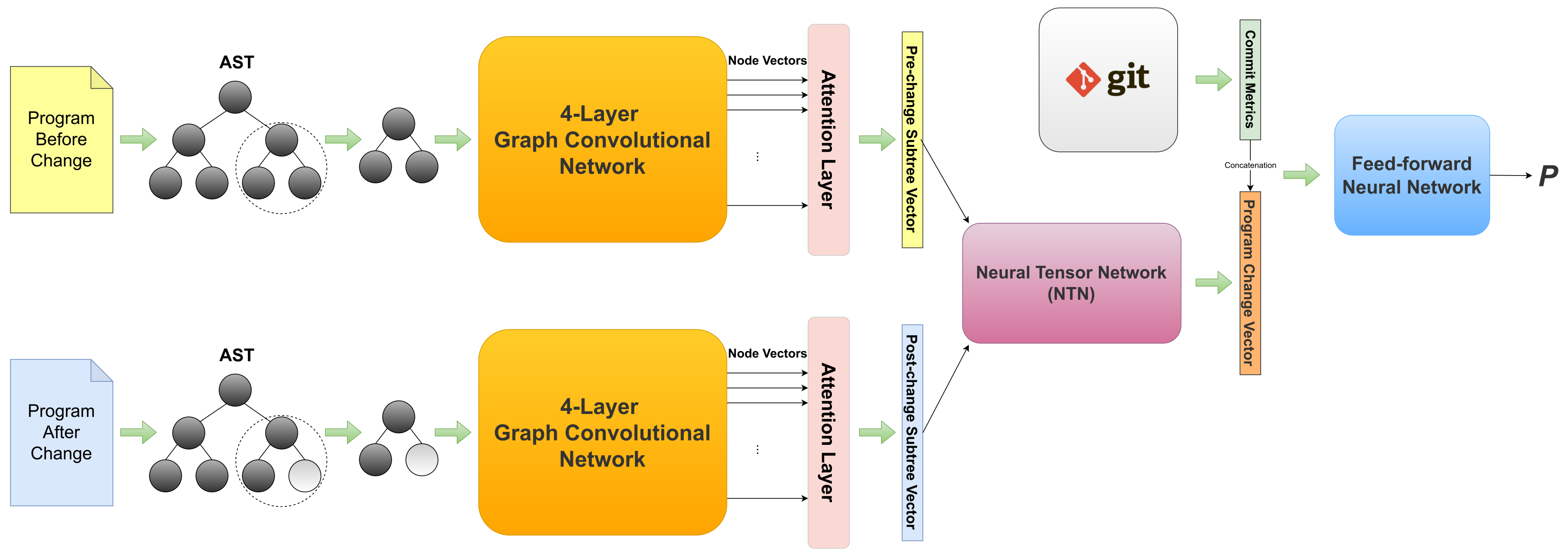

Program Before Change
AST
4-Layer Graph Convolutional Network
Node Vectors
Attention Layer
Pre-change Subtree Vector
Program After Change
AST
4-Layer Graph Convolutional Network
Node Vectors
Attention Layer
Post-change Subtree Vector
Neural Tensor Network (NTN)
git
Commit Metrics
Concatenation
Program Change Vector
Feed-forward Neural Network
P

Figure 5.1: Overview of the JITGNN workflow

pass in the JITGNN training passes all the aforementioned components and after comparing the output probability with the actual label of the given sample and calculating 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.

In the following, we explain each step in detail.

5.2 Change Subtrees & Commit Graphs

The main part of JITGNN is done by the two graph neural networks (GNNs) that analyze the content of the changes that were made in a commit. This part requires the commit to be represented with graphs. Essentially, a commit consists of multiple changed files. In this work, we concentrated on the bugs caused by changing source code files rather than all types of files. Therefore, in our context, a commit consists of multiple source codes. In JITGNN, source codes are converted to ASTs based on their context-free grammar and the ASTs are used to make the graphs that are fed into the GNNs.

Depending on the program, ASTs can be very large while the change in the source code often involves a small part of the source code. Considering the AST of a source code before a change and the AST of the same source code after the change, usually, the proportion of AST nodes that are involved in the change to all the nodes is 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 nodes that are changed between the pre-change AST and the post-change AST. 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 Subtree*s. Each source code that is changed in a commit has two change subtrees: one *Pre-change Subtree* and one *Post-change Subtree.*

As we mentioned above, 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 and not on single source code files. Thus, for every commit, we combine all the pre-change subtrees of source codes changed in a commit and form one big disconnected graph called *Pre-change Commit Graph.* Similarly, for post-change subtrees, we make a *Post-change Commit Graph.* This does not affect the graphs and only associates each commit with two graphs to make the prediction process straightforward.

5.3 Node Features

Each layer in GNNs, generally, 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*:

- Nodes in program ASTs have *Node Type*s (grammar non-terminal) and may have *Node Value*s (grammar terminal). In JITGNN, node types (and node values if exist) form text strings that we refer to as *Node Tokens*.
- As we discussed in Section 5.2 we extracted change subtrees from ASTs. The nodes included in these subtrees are the changed nodes, their parents, their children, and their siblings. Accordingly, 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* and it is 1 if the node is changed and 0 otherwise.

Node tokens that we explained above 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 [64, 1, 8, 21]. Although these studies present sophisticated techniques to transform code tokens to vectors, in our experiments, we observed that they do not positively contribute to the final prediction of JITGNN. On the other hand, we were able to achieve good performances with the simple bag-of-words (BoW) technique. Prior JIT bug prediction models that require vectorizing textual data (either code or commit message) have also used BoW [30, 32, 58].

The difference between the performance of simple and complex vectorization techniques in JIT bug prediction 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 bug prediction models mainly use syntactical information of 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.

In JITGNN, we implemented the binary BoW. In binary BoW, node token vectors are made up of 0s and 1s no matter how many times each token is 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.

5.4 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 [76]. 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 embeding of node i at timestep k. The hidden states are computed in the following way:

$$h_i^k = \begin{cases} x_i & k = 0 \\ g(\mu_i^{k-1}, h_i^{k-1}) & \text{otherwise} \end{cases} \tag{5.1}$$

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 summation.

Although the backbone of GNNs is similar, the way 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) [3], the current state and the summation of received hidden states (messages) update the node embedding using GRU units [14], while in a graph convolutional network (GCN) [41] 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 [45, 3, 11, 79]— and comparing its performance in JITGNN with GCN, we picked 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 4 layers (timesteps) in a GCN means that at each forward pass during the training, node hidden states (messages) are sent 4 times throughout the graph and nodes update their hidden states 4 times by the received messages. In other words, the information of node i at each iteration, directly goes to the neighbor nodes (the nodes it has direct edges to) 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 and this information 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 4 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 have a huge impact on 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 nodes 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, 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 [41], we normalized the adjacency matrices of the graphs and the initial node features to train GCNs.

5.5 Attention Mechanism

The goal in 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 each change subtrees. 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. 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 that is adopted in GNNs is to use a dummy supernode [66, 44]. The idea in this technique is to add a virtual node to the graph that is 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 will be updated like all other nodes and because the weights are adjusted during the training, this node will contain more information from the nodes that are more important in the graph.

In JITGNN, we implemented an approach that is 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 first was used in Nair et al. [54]. Formally we have:

$$W_i' = \sigma(h_i.tanh(\frac{1}{|V|}\Sigma_{i \in V} h_i.W_i)) \tag{5.2}$$

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 [25]. As the attention layer is jointly trained, W's are adjusted to be used 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 pre-change subtree and post-change subtree, *Pre-change Subtree Embedding* and *Post-change Subtree Embedding* respectively.

5.6 Neural Tensor Network

After obtaining the global embeddings of the pre-change subtree and the post-change subtree from the two GCNs, the embedding vectors should be aggregated. 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 the 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 static operations on the pre-change and post-change subtree embeddings including cosine similarity (to measure how different the vectors are in the hidden space) and simple concatenation. 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 components of JITGNN and performs a comparison between the two input vectors. This layer is based on the neural tensor network (NTN) proposed in Socher et al. [68].

Originally 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:

$$S(e_1, e_2) = f(e_1^T W e_2 + W' \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} + b). \tag{5.3}$$

In Equation 5.3, $S \in \mathbb{R}^k$ 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 $i = 1, ..., k$ 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 that is given to an activation function to produce the relationship vector S. The activation function f that is used in JITGNN is *ReLU* [55].

5.7 Commit Metrics

From the early research on JIT bug prediction, commit metrics have been parts of the many JIT prediction models up to this point. They have always had a positive effect on the performance of the prediction. For instance, JITLine [58], the state-of-the-art JIT bug prediction model uses commit metrics along with the added lines and removed lines. In our experiments, we isolated the text part of JITLine and discarded the commit metrics to see how effective the commit metrics are and we observed that the performance of JITLine without commit metrics drops (more on this in Chapter 6). 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 that were used by Kamei et al. [36]. The list and the description of these metrics are shown in Table 3.1. These metrics are categorized into 4 classes: *Size*, *Diffusion*, *History*, *Experience*. In JITGNN, each commit metric is considered as a feature, and the set of commit metrics shown in Table 3.1 forms a feature vector for each commit. This vector is simply 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 combination of the commit metrics and the relationship vector serves as the ultimate change vector of the commit because it contains information both 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).

5.8 Feed-forward Neural Network

In the last step, the ultimate change vector generated by combining the commit metrics and the relationship vector should be used to produce the probability of the given commit being bug-inducing. To this end, 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 that we used in JITGNN is the *Sigmoid* function because we wanted JITGNN to output a probability rather than classifying 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 it is that the sample is a bug-inducing commit. The generated probability along with the actual label of the sample (1 for bug-inducing and 0 for clean) are given to the *Binary Cross-Entropy (BCE)* function to compute the loss for the sample. For actual bug-inducing commits, the higher the generated probability is, the lower is the loss. 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 parameters of the model 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 in a way to reduce the loss in future iterations. The optimizer we used in JITGNN is *Adam* [40].

Chapter 6

Experiments & Results

In this section, we explain our approaches to answering the research questions, the experiments, their setups, and the results.

6.1 Data Preparation

In Chapter 4, we briefly introduced the datasets that we used to answer our research questions. Both OpenStack and ApacheJIT datasets are sets of commits with their corresponding commit metrics. To prepare them for training JITGNN, we take the following steps.

6.1.1 JITGNN Compatibility

As we discussed in Chapter 5, JITGNN converts the commits to graphs made up of abstract syntax trees (ASTs) of the source codes that are changed in the commit. Accordingly, the commits given to JITGNN must include at least one program source code. Also, 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 and deleted source codes or newly added ones should not be included in the data.

To filter out commits according to the two rules mentioned above, we removed the commits that did not have at least one compatible source code file. In other words, the commits in which all the files are non-source-code or all source codes are deleted or newly added were removed. These rules are already covered in ApacheJIT but not all the commits in the OpenStack dataset comply with these rules. Therefore, we removed them from the OpenStack dataset. We refer to this version of the OpenStack dataset as *OpenStack (Ours)*. This dataset has 10,196 commits out of which 1,474 are bug-inducing (Table 6.1). For RQ1, we used the original OpenStack dataset to replicate the results of existing JIT models. In RQ2, to do a fair comparison between models, we replaced the original OpenStack with OpenStack (Ours).

6.1.2 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 bug prediction models should be used to notify developers of potential risks in the changes they have recently submitted. In this scenario, JIT models that are 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 in a timewise manner. 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. [30], [32], and Pornprasit and Tantithamthavorn [58] 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 of 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 size of the data of the last 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 size of the data 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 remove clean commits randomly and removed them in a way that in every year of the training data, we have 50% clean commits and 50% bug-inducing commits. The benefits of this undersampling are twofold. First, we reduce the data to use less computing resources. Secondly, by using a balanced training set we can study how JIT models perform when the class imbalance problem does not exist.

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 reflects a real-life scenario. Table 6.1 shows the statistics of the data that is used in our experiments.

Table 6.1: 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

6.2 Comparison Models

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

6.2.1 Naive

To have a better understanding of the results that the following models achieve, we used a naive classifier. To this end, it is common to employ a random guessing model. Since the 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 assigns 1 to each sample in the test set with a probability of r and assigns 0 with a probability of $1 - r$.

In our version of the OpenStack 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.

6.2.2 Baseline

Although the goal in McIntosh and Kamei [47] is not to propose a JIT bug prediction model, to conduct their experiments, they follow Zhou and Mockus [80], Morales et al. [51], and Mcintosh et al. [48] and employ multiple regression modeling with some techniques to capture the nonlinear relationship between the features. We used this model as our baseline model. The replication package of the study including the regression model is publicly available.

We studied their 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 is fitted in short-term and long-term settings due to the nature of the study in McIntosh and Kamei [47]. 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 chapter, we refer to the multiple regression model as the *Baseline*.

6.2.3 State-of-the-art

Lately, Pornprasit and Tantithamthavorn [58] presented a JIT bug prediction approach called *JITLine*. In their paper, they compare the performance of JITLine with CC2Vec [32], DeepJIT [30], and EALR [36] and 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 bag-of-words (BoW) representation of the changed code. 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. The scripts of JITLine are also available.

As we mentioned above, JITLine uses the added and deleted code tokens in commits. The JITLine replication 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 6.1). However, ApacheJIT does not have the information of 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.

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.

6.3 Evaluation Metrics

To evaluate the discriminatory performance of JITGNN and the comparison models we introduced in the previous part, we selected a combination of threshold-dependent and threshold-independent evaluation metrics that are commonly used in the literature.

- *AUC*: AUC is the area under the receiver operating characteristic (ROC) curve. 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. $Precision = \frac{TruePositive}{TruePositive+FalsePositive}$. 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.

- *Recall*: Recall measures the ability of a classifier not to miss positive samples. $Recall = \frac{TruePositive}{TruePositive+FalseNegative}$. Recall is a threshold-dependent measure and similar to precision, we set its threshold to 0.50.

- F_1 *score*: F_1 score is the harmonic mean of precision and recall. $F_1 score = \frac{2.Precision.Recall}{Precision+Recall}$. F_1 score is also threshold-dependent and we set the threshold to 0.50.

Note that the threshold-dependent metrics (F_1 score, Precision, and Recall) 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 bug prediction models and can handle the class imbalance problem [36, 77, 22, 47, 30, 32, 58]. The reason we used threshold-dependent metrics along with a threshold-independent one is that ultimately, a potential tool that is 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 study the evaluations for a range of different thresholds.

6.4 Experiments & Results

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

Approach: As we explained in the previous part, we studied the replication packages of the baseline and JITLine. After doing the required preparation discussed earlier, we ran the scripts on the original OpenStack dataset to compare the results with the results reported in their respective papers. Table 6.2 shows the results of the replication.

As for the baseline, the results in McIntosh and Kamei [47] are technically for a different experiment but the long-term experiment they did in the six-month setting for their RQ1 is related to our study. The result we obtained from the scripts was 0.81 AUC and 0.09 Brier (Brier score is not presented in Table 6.2). Originally, they report an AUC of 0.72 AUC and 0.11 Brier. The difference between our results can be attributed to the difference between the training and the test sets as we used JITLine's training and test sets, which are slightly different from the training and test sets in McIntosh and Kamei [47] in this particular setting (long-term six-month setting); therefore, we cannot compare our results with theirs. For this regression model, we also calculated the precision, recall, and F_1 score evaluation metrics, which are not reported in the original paper.

Replicating JITLine, on the other hand, leads to almost the same results as the original paper. Running JITLine on our machines achieves the same AUC as the one reported in the paper and slightly better F_1 score, precision, and recall (originally F_1 score=0.33, precision=0.43, and recall=0.26). After carefully understanding the JITLine scripts, we did additional experiments with the JITLine architecture to see how each component in JITLine affects the performance. In this regard, we experimented with three variants of JITLine:

- *JITLine w/o SMOTE w/o Metrics*: In this variant, we discarded the SMOTE oversampling module and the commit metrics. 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.
- *JITLine w/o Metrics*: In this variant, the commit metric features are excluded from the JITLine original feature vectors that are fed to the random forest model.
- *JITLine w/o SMOTE*: In this variant of JITLine does not have the SMOTE oversampling module.

The results of running these variants on the original OpenStack dataset are shown in Table 6.2 along with the results of replicating the original JITLine. These results 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, precision, and recall drop without oversampling. This observation means that the impact of oversampling is more on the threshold beyond which the model classifies a sample as bug-inducing or clean. In other words, the more balanced the training data is, the better the 0.50 threshold works as a distinguishing point.

Table 6.2: 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 w/o SMOTE w/o Metrics	0.81	0.11	0.37	0.06
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	0.83	0.37	0.47	0.30

Result: We were able to successfully replicate the baseline model and JITLine results. The results are in line with the ones reported in their respective papers. We also made three variants of JITLine to further study the performance of JITLine in the presence or absence of the SMOTE module and the commit metric features.

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

Approach: To answer this research question, we ran the baseline, JITLine (and its variants that we introduced above), and JITGNN on the OpenStack (Ours) and ApacheJIT data we prepared. The results are shown in Table 6.3. Similar to JITLine, we repeated the experiments with variants of JITGNN with different architecture configurations:

- *JITGNN w/o Metrics*: Similar to *JITLine w/o Metrics*, we discarded the commit metrics from the ultimate change vector and fed the program change embedding that is 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 neural tensor network (NTN) to output a single vector that captures the relationship between 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. [66] and Li et al. [44]). Also, instead of NTN, the program change embedding is created by concatenating the global graph embedding of the pre-change and post-change subtrees.

The results of the experiments explained above 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, which means it is guessing the labels randomly. In our version of the OpenStack 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 and recall but the precision of JITLine is marginally higher (0.35 vs 0.34).

In ApacheJIT, again, JITGNN and JITLine share the first place in terms of AUC by achieving an AUC of 0.81 while the AUC of the baseline is 0.78. However, the baseline performs better in terms of the F_1 score and precision. As for the recall, JITLine and its variants achieve highest recall scores, 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

Table 6.3: JIT bug prediction model results

	OpenStack (Ours)			
	AUC	**F_1 score**	**Precision**	**Recall**
Naive	0.50	0.16	0.15	0.17
Baseline	0.78	0.14	0.35	0.08
JITLine w/o SMOTE w/o Metrics	0.78	0.10	0.39	0.06
JITLine w/o Metrics	0.76	0.38	0.39	0.38
JITLine w/o SMOTE	**0.79**	0.11	0.41	0.06
JITLine	**0.79**	0.35	**0.45**	0.28
JITGNN w/o Metrics	0.76	**0.43**	0.35	**0.54**
JITGNN Special Token	0.70	0.32	0.28	0.35
JITGNN Supernode - Concatenation	0.75	0.40	0.34	0.49
JITGNN	**0.79**	0.35	0.44	0.28

	ApacheJIT			
	AUC	**F_1 score**	**Precision**	**Recall**
Naive	0.49	0.28	0.19	0.51
Baseline	0.78	**0.69**	**0.73**	0.65
JITLine w/o SMOTE w/o Metrics	0.79	0.38	0.25	0.81
JITLine w/o Metrics	0.79	0.39	0.26	**0.82**
JITLine w/o SMOTE	**0.81**	0.50	0.36	0.81
JITLine	**0.81**	0.49	0.35	0.80
JITGNN w/o Metrics	0.80	0.48	0.36	0.75
JITGNN Special Token	0.75	0.35	0.30	0.39
JITGNN Supernode - Concatenation	0.77	0.40	0.34	0.49
JITGNN	**0.81**	0.50	0.37	0.78

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.

Table 6.4 provides insight into how the three models (the baseline, JITLine, and JITGNN) 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.

With these definitions, in Table 6.4, *ModelName Wrong* corresponds to the times when model *ModelName* has misclassified the test samples and *ModelName Right* corresponds to the times when model *ModelName* has correctly classified 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.

Table 6.4: 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	22	18	28
JITGNN Right	18	26	24	804

(b) ApacheJIT

	JITLine Wrong Baseline Wrong	**JITLine Wrong Baseline Right**	**JITLine Right Baseline Wrong**	**JITLine Right Baseline Right**
JITGNN Wrong	1,344	216	91	563
JITGNN Right	572	287	251	4,191

Table 6.4 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 the OpenStack dataset, the numbers are close. In the ApacheJIT dataset, however, we can observe some differences. JITGNN predicts the labels of 572 test samples correctly while the other two models predict them incorrectly. On the other hand, the number of samples JITLine predicts correctly and the other two models misclassify is 91. This number for the baseline is 216. In other words, JITGNN is able to correctly label samples when the other two models are not able to more than the other two models.

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

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 6.5 shows the results of these experiments.

Predictably, for low thresholds, the recall is high because many samples are classified as positive. Going toward higher thresholds, recall decreases, and precision increases generally. The exceptions are where the classifier is giving low probabilities to true positive samples. In these cases, the samples whose probability of positiveness is beyond the threshold are not actually positive and it drops the precision. For example, in Table 6.5a, the precision of the baseline drops at $threshold = 0.60$ and reaches 0 for the subsequent thresholds.

Result: For this research question, overall, we can say that JITGNN performs marginally better than the baseline and its performance is comparable to the state-of-the-art (JITLine). The study of 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 their users.

6.4.3 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. 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 ratios of bug-inducing commits in all these training sets were 0.50 while this ratio was 0.19 in the test set.

The results of these experiments are shown in Table 6.6. Similar to previous research questions, We report AUC, F_1 score, precision, and recall of the models.

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.

Result: 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.

Table 6.5: Evaluations of threshold-dependent metrics for thresholds in [0.20, 0.70] with increaments of 0.05

(a) OpenStack (Ours)

		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 6.6: JITGNN performance on different sizes of ApacheJIT training set

Size of ApacheJIT Training Data	**AUC**	**F_1 score**	**Precision**	**Recall**
4,483 (10%)	0.7768	0.4602	0.3366	0.7350
8,967 (20%)	0.7787	0.4617	0.3244	0.8004
13,450 (30%)	0.7767	0.4995	0.4449	0.5694
17,934 (40%)	0.7814	0.4945	0.4092	0.6246
22,417 (50%)	0.7988	0.4905	0.3550	0.7934
26,900 (60%)	0.7949	0.4984	0.3763	0.7383
31,384 (70%)	0.8018	0.4676	0.3260	0.8269
35,867 (80%)	0.8017	0.5048	0.3797	0.7530
40,351 (90%)	0.8042	0.5036	0.3794	0.7488
44,834 (100%)	0.8087	0.5038	0.3711	0.7844

Chapter 7

Discussion

7.1 JITGNN Performance

In this work, we attempted to include the graph structure of source codes into a Just-In-Time (JIT) bug prediction model by using abstract syntax trees (ASTs) of programs given to a graph neural network (GNN) framework. Our assumption was that by combining the code syntax and semantic information and the change metadata, we can improve the discriminatory performance of JIT models. Our study for RQ2 and the results presented in Section 6 does not support our assumption and the performance of JITGNN is comparable to the existing state-of-the-art JIT model (JITLine).

In this study, we used two datasets. The first dataset is a collection of OpenStack project commits built by McIntosh and Kamei [47]. This dataset —similar to other JIT and generally other forms of bug prediction datasets— is not balanced. The ratio of bug-inducing commits in this dataset after preparation (discussed in Section 6.1) is 0.14 (Table 6.1). The second dataset is a newly created cross-project dataset from commits in 14 Apache projects called ApacheJIT [37]. As we discussed the data preparation process, the training data we used from this dataset was the result of undersampling the majority class (clean commits). This undersampling was done by keeping the same ratio of bug-inducing commits and clean commits in each period (year). Accordingly, the training data (and not the test data) from ApacheJIT was balanced.

By comparing the performances of the three JIT models on these two datasets in Table 6.3, we observe that having a balanced training set results in higher values of threshold-dependent metrics at $threshold = 0.50$. Also, the results in Table 6.5 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.

Another interesting observation in this work was the performance of the cross-lingual JIT-GNN. As we explained the ApacheJIT dataset and its cross-project nature, and considering the results presented in Section 6, JITGNN has an acceptable cross-project performance. In another experiment, we attempted to evaluate 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 (Ours).

The reason we chose the *JITGNN Special Token* was to reduce the size of vocabulary 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 were able to map 49 tokens using the grammars of Java and Python. JITGNN had a poor performance in this experiment (0.65 AUC, 0.13 F_1 score, 0.29 precision, and 0.08 recall).

7.2 Khata

JIT bug prediction models are trained on historical data and their ultimate goal is to notify developers that are submitting changes (commit authors in the context of Git) and warn them that their change is likely to be defect-prone. This gives the developer an opportunity to review the change again and prevent future bugs. Although there has been a substantial amount of work done on JIT models in recent years, not many practical JIT tools have been developed. This sometimes has led to unrealistic experiment settings that are proposed in the literature [70, 60].

To complement our study on JIT models and designing JITGNN, we implemented an open-source practical JIT tool called *Khata* that developers can install on their machines and run after staging the changes[1]. Khata finds the changed files, filters out the files that are newly added or completely removed, and collects the change metrics. Next, Khata loads the JIT model that it is set to, runs the JIT model to predict the probability of the change being bug-inducing, and reports this probability. JIT models in Khata are extensible and anyone can add new JIT models to it. Khata is written in Python and currently includes JITGNN and JITLine as JIT models. The workflow of Khata is shown in Figure 7.1.

Khata outputs a probability instead of classifying the changes into two groups of clean and bug-inducing. The reason is that having a hard threshold to assess changes is not desirable in many software projects. Plus, each project, depending on its context, may choose to set a high or low threshold, respectively to have more confidence in the changes they need to review and to make sure they do not miss any bug-inducing commits. Figure 7.2 and Figure 7.3 show the distribution of probabilities JITGNN and JITLine output on the positive (bug-inducing commits) and negative (clean commits) samples in OpenStack and ApacheJIT test data. These figures show that the thresholds that are chosen may differ from one project to

[1]Khata is available at: https://github.com/uw-swag/khata

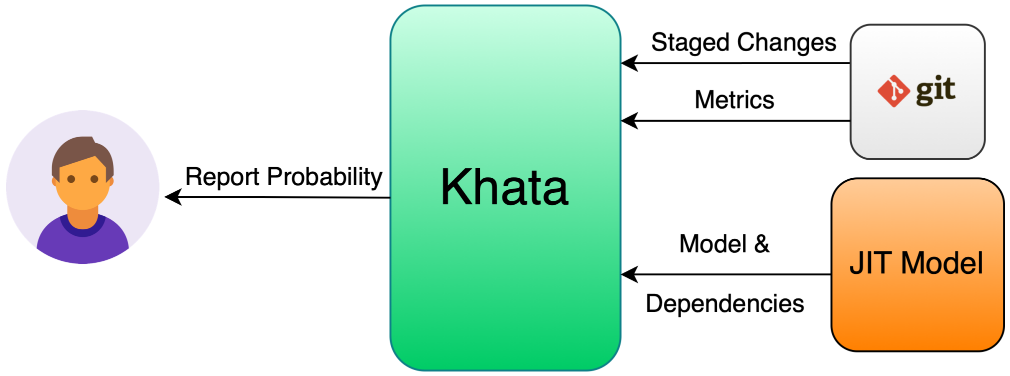
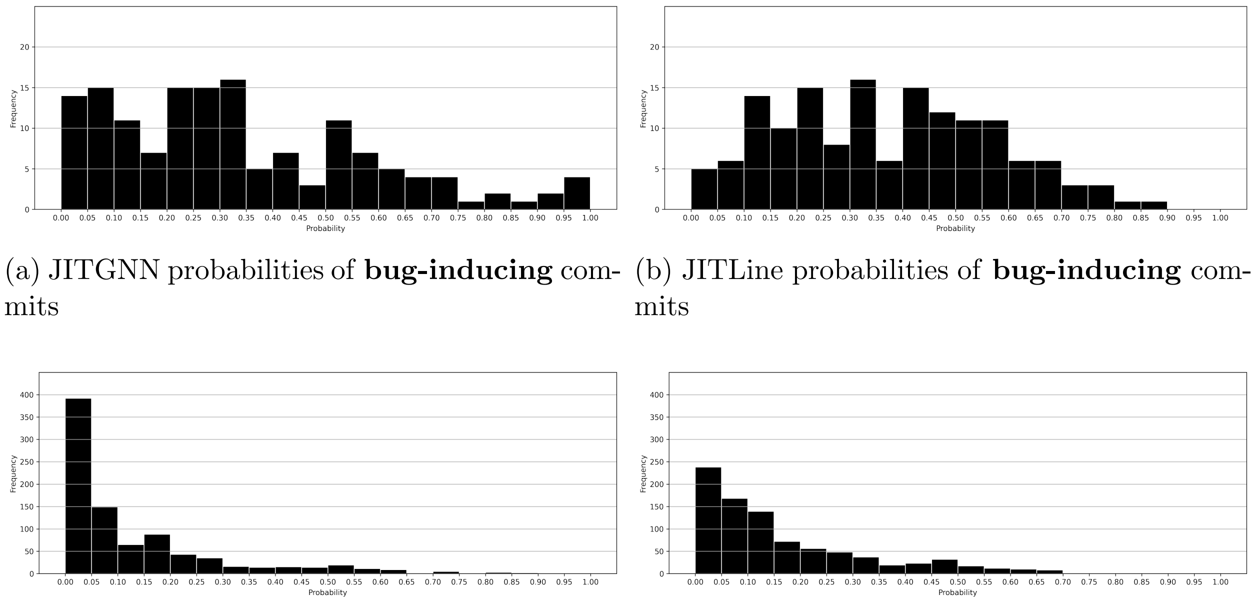


Figure 7.1: Workflow of Khata



(a) JITGNN probabilities of **bug-inducing** commits (b) JITLine probabilities of **bug-inducing** commits

(c) JITGNN probabilities of **clean** commits (d) JITLine probabilities of **clean** commits

Figure 7.2: The distribution of probabilities generated by JITGNN and JITLine on the **OpenStack** dataset

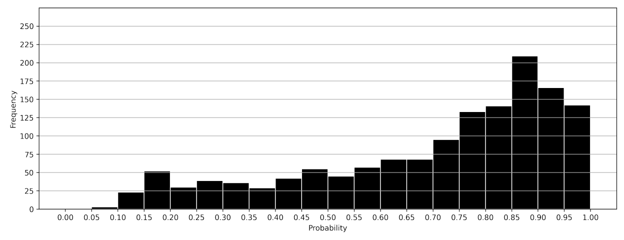
another. Also, according to these figures, even picking a low threshold does not guarantee that all bug-inducing commits will be detected.

We should mention that Khata is not evaluated in practice by developers yet.

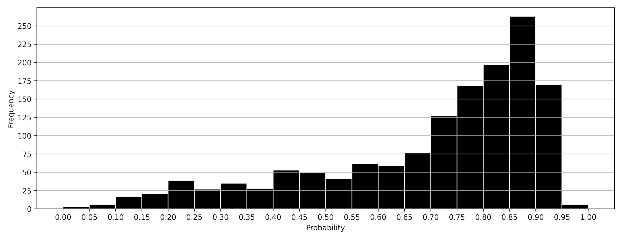
7.3 Threats to Validity

7.3.1 Construct Validity

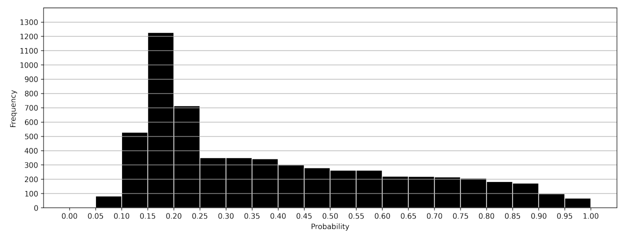
Threats to construct validity are related to how proper our inferences are in this study based on the experiments and the results. In this work, we studied the application of graph neural



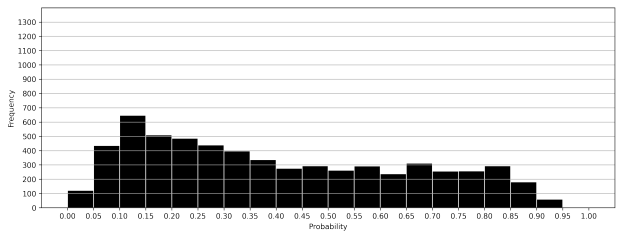
(a) JITGNN probabilities of **bug-inducing** commits



(b) JITLine probabilities of **bug-inducing** commits



(c) JITGNN probabilities of **clean** commits



(d) JITLine probabilities of **clean** commits

Figure 7.3: The distribution of probabilities generated by JITGNN and JITLine on the **ApacheJIT** dataset

networks (GNNs) and abstract syntax trees (ASTs) in the context of Just-In-Time (JIT) bug prediction. 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 graph neural networks (GNNs) and abstract syntax trees (ASTs) as GNNs may not capture the structure of the code represented in ASTs well in the context of JIT bug prediction.

Furthermore, recently, Rodriguez-Perez et al. [65] introduced a group of bugs that are caused by external factors such as changes in requirements or changes in APIs. They call this group of bugs *Extrinsic Bugs*. These bugs and their causes cannot be found using the SZZ algorithm that is employed to construct both datasets we used in this work. The authors suggest researchers exclude this type of bug-inducing changes from the data they are feeding to JIT models. This approach improves the performance of JIT models but on the other hand, JIT models are not able to identify these bugs. Therefore, the results of this study cannot be applied in the context of extrinsic bugs.

7.3.2 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 [47] and used in its subsequent JIT bug prediction studies [30, 32, 58] and the other dataset is a newly created JIT dataset called ApacheJIT [37]. ApacheJIT is a cross-project JIT dataset that has the historical change data from 14 popular Apache projects. ApacheJIT is a large dataset that contains more than 100,000 commits after an extensive filtering process suggested by da Costa et al. [15] and McIntosh and Kamei [47].

Although ApacheJIT is a large dataset containing commits from a number of different projects, still there is an external validity threat that we cannot necessarily generalize the results we obtained in our experiments to other software projects. Also, the main 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 grammar of the language.

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

7.3.3 Internal Validity

Threats to the internal validity of our results relate to the uncontrolled factors that might have had an impact on our results. Both the OpenStack and the ApacheJIT datasets that are used in this work are constructed based on the SZZ algorithm [85]. SZZ is the most widely used algorithm to automatically extract the bug-inducing commits in a software Git repository. Despite the extensive use of SZZ, it has its limitations [15, 56, 57]. To mitigate these limitations, da Costa et al. [15] and McIntosh and Kamei [47] propose a set of filtering steps to remove suspicious bug-inducing changes. Both datasets that were used in this work were built by adhering to these filtering steps.

As JITGNN is a deep graph neural network and requires computing a large number of parameters in each layer, we conducted the training of 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 problem, 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 mostly less than 0.01 AUC with some exceptions that had a 0.01 difference (± 0.01 AUC).

Other internal validity threats in our study are the scripts that were 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 [47] 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 [19]. GumTreeDiff is a source code differencing tool that also extracts the ASTs of programs written in the supporting languages and identifies the differences between the ASTs of two source codes.

Chapter 8

Future Work & 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 on 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 that are 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 compared the performance of JITGNN against multiple regression modeling [47] as the baseline and JITLine [58] 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 [47] and the other one is a newly built JIT dataset called ApacheJIT [37]. Our results show that JITGNN performance is comparable to the state-of-the-art. 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 but the increase in evaluation metrics are not very sharp and even with 10% of the ApacheJIT training set, JITGNN achieves acceptable performance.

Finally, we implemented an open-source JIT tool for developers called Khata. Khata analyzes the staged changes in a Git working directory and reports the probability that the change is bug-inducing. Currently, Khata has two JIT models: JITGNN and JITLine. However, Khata is extensible and users can add new JIT models to it. Khata is not evaluated by developers. Further study is needed to evaluate how effective Khata is to prevent bug-inducing commits. Also, the convenience of using and extending JIT models in Khata should be investigated.

Continuing this research thread, potentially JITGNN can be improved by including more graphical program information. Data flow 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 [81]. One challenge in JITGNN training is the training time. We reduced the size of the ASTs given to GCNs in JITGNN by removing the nodes that are not related to the changed nodes. However, more reduction is needed to increase the speed of training.

Another study that can be derived from this work is the analysis of the commits that are correctly and incorrectly classified by JITGNN and compare them with the classification that is done by other models to get more insight into what types of bug-inducing commits JITGNN is able to detect and what bug-inducing commits it misses.

References

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 38–49, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786849. URL https://doi.org/10.1145/2786805.2786849.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001, 2016. URL http://arxiv.org/abs/1602.03001.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017. URL http://arxiv.org/abs/1711.00740.

[4] Md Zahangir Alom, Tarek M. Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Mahmudul Hasan, Brian C. Van Essen, Abdul A. S. Awwal, and Vijayan K. Asari. A state-of-the-art survey on deep learning theory and architectures. *Electronics*, 8(3), 2019. ISSN 2079-9292. doi: 10.3390/electronics8030292. URL https://www.mdpi.com/2079-9292/8/3/292.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 404–419, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192412.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *SIGPLAN Not.*, 53(4):404–419, jun 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192412. URL https://doi.org/10.1145/3296979.3192412.

[7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *ArXiv*, abs/1808.01400, 2019.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290353. URL https://doi.org/10.1145/3290353.

[9] Erik Arisholm and Lionel C. Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, page 8–17, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595932186. doi: 10.1145/1159733.1159738. URL https://doi.org/10.1145/1159733.1159738.

[10] Jayme Garcia Arnal Barbedo. Impact of dataset size and variety on the effectiveness of deep learning and transfer learning for plant disease classification. *Computers and Electronics in Agriculture*, 153:46–53, 2018. ISSN 0168-1699. doi: https://doi.org/10.1016/j.compag.2018.08.013. URL https://www.sciencedirect.com/science/article/pii/S0168169918304617.

[11] Marc Brockschmidt. GNN-FiLM: Graph neural networks with feature-wise linear modulation. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1144–1152. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/brockschmidt20a.html.

[12] Deyu Chen, Xiang Chen, Hao Li, Junfeng Xie, and Yanzhou Mu. Deepcpdp: Deep learning based cross-project defect prediction. *IEEE Access*, 7:184832–184848, 2019. doi: 10.1109/ACCESS.2019.2961129.

[13] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.*, 30(3), apr 2021. ISSN 1049-331X. doi: 10.1145/3436877. URL https://doi.org/10.1145/3436877.

[14] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *SSST@EMNLP*, 2014.

[15] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43 (7):641–657, 2017. doi: 10.1109/TSE.2016.2616306.

[16] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. A deep tree-based model for software defect prediction. *CoRR*, abs/1802.00921, 2018. URL http://arxiv.org/abs/1802.00921.

[17] Shi Dong, Ping Wang, and Khushnood Abbas. A survey on deep learning and its applications. *Computer Science Review*, 40:100379, 2021. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2021.100379. URL https://www.sciencedirect.com/science/article/pii/S1574013721000198.

[18] Khaled El Emam, Walcelio Melo, and Javam C Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of systems and software*, 56(1): 63–75, 2001.

[19] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014. doi: 10.1145/2642937.2642982. URL http://doi.acm.org/10.1145/2642937.2642982.

[20] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E. Hassan, and Shanping Li. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 47(8):1559–1586, 2021. doi: 10.1109/TSE.2019.2929761.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

[22] Wei Fu and Tim Menzies. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 72–83, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106257. URL https://doi.org/10.1145/3106237.3106257.

[23] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 172–181, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597075. URL https://doi-org.proxy.lib.uwaterloo.ca/10.1145/2597073.2597075.

[24] Jiri Gesi, Jiawei Li, and Iftekhar Ahmed. *An Empirical Examination of the Impact of Bias on Just-in-Time Defect Prediction.* Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450386654. URL https://doi.org/10.1145/3475716.3475791.

[25] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL https://proceedings.mlr.press/v9/glorot10a.html.

[26] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000. doi: 10.1109/32.859533.

[27] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 78–88, 2009. doi: 10.1109/ICSE.2009.5070510.

[28] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554, 07 2006. ISSN 0899-7667. doi: 10.1162/neco.2006.18.7.1527. URL https://doi.org/10.1162/neco.2006.18.7.1527.

[29] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 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)*, pages 34–45, 2019. doi: 10.1109/MSR.2019.00016.

[30] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 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)*, pages 34–45, 2019. doi: 10.1109/MSR.2019.00016.

[31] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 518–529, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380361.

[32] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 518–529, 2020.

[33] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289, 2013. doi: 10.1109/ASE.2013.6693087.

[34] Justin M. Johnson and Taghi M. Khoshgoftaar. Survey on deep learning with class imbalance. *Journal of Big Data*, 6:1–54, 2019.

[35] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010. doi: 10.1109/ICSM.2010.5609530.

[36] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi: 10.1109/TSE.2012.70.

[37] Hossein Keshavarz and Meiyappan Nagappan. Apachejit: A large dataset for just-in-time defect prediction, 2022. arXiv:2203.00101.

[38] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE'07)*, pages 489–498, 2007. doi: 10.1109/ICSE.2007.66.

[39] Sunghun Kim, E. James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008. doi: 10.1109/TSE.2007.70773.

[40] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[41] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[42] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. *Improved Code Summarization via a Graph Neural Network*, page 184–195. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450379588. URL https://doi.org/10.1145/3387904.3389268.

[43] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328, 2017. doi: 10.1109/QRS.2017.42.

[44] Junying Li, Deng Cai, and Xiaofei He. Learning graph-level representation for drug discovery. *arXiv preprint arXiv:1709.03741*, 2017.

[45] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05493.

[46] C. Manjula and L. Z. Florence. Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Computing*, 22:9847–9863, 2018.

[47] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, 2018. doi: 10.1109/TSE.2017.2693980.

[48] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Softw. Engg.*, 21(5):2146–2189, oct 2016. ISSN 1382-3256. doi: 10.1007/s10664-015-9381-9. URL https://doi.org/10.1007/s10664-015-9381-9.

[49] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 107–116, 2010. doi: 10.1109/CSMR.2010.18.

[50] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000. doi: 10.1002/bltj.2229.

[51] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 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)*, pages 171–180, 2015. doi: 10.1109/SANER.2015.7081827.

[52] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 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*, pages 181–190, 2008. doi: 10.1145/1368088.1368114.

[53] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292, 2005. doi: 10.1109/ICSE.2005.1553571.

[54] Aravind Nair, Avijit Roy, and Karl Meinke. 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, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375801. doi: 10.1145/3382494.3410675. URL https://doi.org/10.1145/3382494.3410675.

[55] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

[56] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 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)*, pages 380–390, 2018. doi: 10.1109/SANER.2018.8330225.

[57] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. Revisiting and improving szz implementations. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019. doi: 10.1109/ESEM.2019.8870178.

[58] Chanathip Pornprasit and Chakkrit Tantithamthavorn. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 369–379, 2021.

[59] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. A survey on deep learning:

Algorithms, techniques, and applications. *ACM Comput. Surv.*, 51(5), sep 2018. ISSN 0360-0300. doi: 10.1145/3234150. URL https://doi.org/10.1145/3234150.

[60] Lei Qiao and Yan Wang. Effort-aware and just-in-time defect prediction with neural network. *PloS one*, 14(2):e0211359, 2019.

[61] Lei Qiao, Xuesong Li, Qasim Umer, and Ping Guo. Deep learning based software defect prediction. *Neurocomputing*, 385:100–110, 2020. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2019.11.067.

[62] Santosh Singh Rathore and Atul Gupta. Investigating object-oriented design metrics to predict fault-proneness of software modules. In *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, pages 1–10, 2012. doi: 10.1109/CONSEG.2012.6349484.

[63] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 111–124, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2677009. URL https://doi.org/10.1145/2676726.2677009.

[64] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". *SIGPLAN Not.*, 50(1):111–124, jan 2015. ISSN 0362-1340. doi: 10.1145/2775051.2677009. URL https://doi.org/10.1145/2775051.2677009.

[65] Gema Rodriguez-Perez, Meiyappan Nagappan, and Gregorio Robles. Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. doi: 10.1109/TSE.2020.3021380.

[66] F Scarselli, M Gori, Ah Chung Tsoi, M Hagenbuchner, and G Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 1(20):61–80, 2009.

[67] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. 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, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316149. doi: 10.1145/2393596.2393670. URL https://doi-org.proxy.lib.uwaterloo.ca/10.1145/2393596.2393670.

[68] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[69] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of*

Software Engineering, ESEC/FSE 2018, page 908–911, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3264598. URL https://doi.org/10.1145/3236024.3264598.

[70] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108, 2015. doi: 10.1109/ICSE.2015.139.

[71] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108, 2015. doi: 10.1109/ICSE.2015.139.

[72] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. Convolutional neural networks over control flow graphs for software defect prediction. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 45–52, 2017. doi: 10.1109/ICTAI.2017.00019.

[73] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308, 2016. doi: 10.1145/2884781.2884804.

[74] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 46(12): 1267–1293, 2020. doi: 10.1109/TSE.2018.2877612.

[75] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 46(12): 1267–1293, 2020. doi: 10.1109/TSE.2018.2877612.

[76] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32:4–24, 2019.

[77] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26, 2015. doi: 10.1109/QRS.2015.14.

[78] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 157–168, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950353. URL https://doi.org/10.1145/2950290.2950353.

[79] Jiaojiao Yu, Kunsong Zhao, Jin Liu, Xiao Liu, Zhou Xu, and Xin Wang. Exploiting gated graph neural network for detecting and explaining self-admitted technical debts. *Journal of Systems and Software*, 187:111219, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2022.111219.

[80] Minghui Zhou and Audris Mockus. Does the initial environment impact the future of developers. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 271–280, 2011. doi: 10.1145/1985793.1985831.

[81] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *NeurIPS*, pages 10197–10207, 2019.

[82] Xiangxin Zhu, Carl Vondrick, Charless C Fowlkes, and Deva Ramanan. Do we need more training data? *International Journal of Computer Vision*, 119(1):76–92, 2016.

[83] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 531–540, 2008. doi: 10.1145/1368088.1368161.

[84] Ayşe Nur Çayır and Tuğba Selcen Navruz. Effect of dataset size on deep learning in voice recognition. In *2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, pages 1–5, 2021. doi: 10.1109/HORA52670.2021.9461395.

[85] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595931236. doi: 10.1145/1083142.1083147. URL https://doi.org/10.1145/1083142.1083147.