

Graph-Based and Anomaly Detection Learning Models for Just-in-Time Defect Prediction

A Dissertation Presented for the
Doctor of Philosophy
Degree

The University of Tennessee, Knoxville

Aradhana Soni

May 2024

© by Aradhana Soni, 2024
All Rights Reserved.

Acknowledgments

I would like to thank my advisors, Dr. Anahita Khojandi and Dr. James Ostrowski for guiding me on this educationally enriching journey and supporting me throughout. I would also like to thank my committee members, Dr. Kamesh Madduri and Dr. Bing Yao for their continued help. I want to thank my graduate school colleagues Rupam Dey and Jeremy Watts for their valuable feedback and support during my thesis. I would also like to acknowledge the Department of Industrial and Systems Engineering at University of Tennessee, Knoxville and all the professors in the department. In conclusion, I wish to convey my deep appreciation to my husband, who has consistently offered unwavering support and ongoing encouragement during the journey of researching and composing my thesis. I also extend my heartfelt thanks to my beloved daughter, whose support has been invaluable. Furthermore, I am grateful to my parents, sisters and friends for their unwavering belief in me, as well as their enduring support and motivation.

This research was conducted with the support and funding provided by grant #2022-244141 from the Silicon Valley Community Foundation's Cisco University Research Program Fund. Their generosity has been instrumental in the success of this research project. I would also like to acknowledge Dr. Kalyan Perumalla for his guidance and support as primary thesis advisor from May 2021 to February 2023.

Abstract

Efficiently identifying and resolving software defects is essential for producing high-quality software. Early and accurate prediction of these defects plays a pivotal role in maintaining software quality. This dissertation focuses on advancing software defect prediction methodologies and practical applications by incorporating graph-based learning techniques and generative adversarial-based anomaly detection techniques.

First, we present a novel approach to software defect prediction by introducing a Graph-based Defect Ratio (GDR). This innovative metric leverages the intricate graph structure that captures the interdependencies among developers, commits, and repositories, offering a promising alternative to standard traditional features. This study highlights the potential for graph-based feature engineering as an alternative or complement to the use of conventional features in defect prediction models, offering a more efficient approach to enhance software quality.

Second, we apply a new Generative Adversarial Network-based technique for detecting anomalies (GAN-AN). We tailor this technique to identify software defects early in the software development process. Our technique, which detects defects as anomalies, is compared against three established anomaly detection methods: autoencoder, isolation forest, and one-class support vector machine (OCSVM). Our investigation reveals that while GAN-AN surpasses existing state-of-the-art methods, the effectiveness of models can differ significantly. This variation in performance metrics implies that the choice of the most suitable model could depend on the particular dataset and the specific objectives of defect detection.

Finally, we explore the efficiency of just-in-time software defect prediction (JIT-SDP) using a novel tri-partite graph structure and Relational Graph Convolutional Networks

(R-GCNs). Our detailed examination of the connections among developers, commits, and repositories deepen our insight into the dynamics of software defects. The study demonstrates that the performance of advanced computational models in defect prediction is closely tied to the specific characteristics of the data and the context of the problem. This research contributes to the ongoing dialogue in software defect prediction, underscoring the importance of an approach that judiciously combines sophisticated computational techniques with more traditional, proven methods.

Table of Contents

1	Graph-Based Feature Engineering for Just-in-Time Software Defect Prediction	1
1.1	Introduction	1
1.2	Background, Gaps, and Contributions	4
1.2.1	Just-in-Time Software Defect Prediction	4
1.2.2	Gaps and Contributions	6
1.3	Data and Methods	7
1.3.1	Data	7
1.3.2	Problem Formulation	8
1.3.3	Models	13
1.3.4	Metrics and Model Evaluation	13
1.3.5	Experiments and Rationale	14
1.4	Results	16
1.4.1	Feature Importance	16
1.4.2	GDR Composition	16
1.4.3	Comparison Against Top Traditional Feature	19
1.4.4	Comparative Evaluation with Composite Feature Sets	19
1.5	Conclusion and Future Work	21
2	Generative Adversarial Network Based Anomaly Detection for Software Defect Prediction	23
2.1	Introduction	23
2.2	Related Work, Gaps and Contributions	26

2.2.1	Background	26
2.2.2	Gaps and Contributions	28
2.3	Methods	29
2.3.1	Data	29
2.3.2	Models	29
2.3.3	Model Evaluation	37
2.4	Numerical Results	38
2.4.1	Within-project testing	38
2.4.2	Cross-project testing	40
2.5	Conclusion and Future Works	42
3	Does Graph Neural Network Improve Software Defect Prediction?	44
3.1	Introduction	44
3.2	Background and Related Work	46
3.2.1	Our contribution	48
3.3	Methods	49
3.3.1	Datasets	49
3.3.2	Problem Formulation	49
3.3.3	Models	56
3.3.4	Evaluation Metrics	59
3.4	Results and Discussion	60
3.5	Conclusion and Future Work	64
	Bibliography	67
	Vita	76

List of Tables

1.1	Traditional Features Characterized by 5 Dimensions	9
1.2	Statistics of the datasets used in our study	10
1.3	Comparison of Model Performances Utilizing GDR, Developer Experience, and Lines of Code as Input Features	18
1.4	Comparison of Model Performances Utilizing GDR vs. Top-Performing Traditional Feature	20
2.1	Statistics of the datasets used in our study	30
2.2	Features Descriptions	31
2.3	Means and Standard Deviations of Key Metrics Across Datasets.	39
3.1	Statistics of the datasets used in our study.	50
3.2	Traditional Features Characterized by 5 Dimensions	51
3.3	GNN performance metrics	61
3.4	Logistic regression performance metrics	62

List of Figures

1.1	The nodes on the left, that is $v_d \in V^D$, represent the developer's label, nodes in the middle column, $v_c \in V^C$, represent the commits, and nodes in the right column, $v_r \in V^R$, represent the repository to which the commit has been made.	11
1.2	This plot offers feature importances based on their influence on the model's predictions. GDR has the highest importance followed by other features.	17
2.1	Overview of the GAN-AN approach, which contains two encoders, a decoder, and discriminator networks, employed within three sub-networks. An autoencoder which is also the generator of the model learns to recreate the original input using an encoder and a decoder. The discriminator is trained to distinguish the real data point from the fake. The second encoder comprises the reconstruction into a latent code $\hat{z}$. The objective function considers three types of losses including contextual, encoder, and adversarial loss.	34
2.2	Isolating an anomalous data point using recursive splitting. The data point X_i is a probable anomaly as only two splits are needed to isolate it.	36
2.3	Comparison of the results over four model approaches: Autoencoder, GAN-AN, isolation forest, and OCSVM. An important observation is that the four metrics: Precision, Recall, F1 score, and AUC-PR are substantially higher than g-means. There are some outliers too especially while fitting the autoencoder model. GAN-AN performs better than isolation forest and OCSVM especially while looking at g-means	41

2.4	Cross-project evaluation metrics: For each dataset, models underwent training and were subsequently assessed on the four other datasets. The PR-AUC and g-mean scores are summarized in the tables provided. Specifically, the matrix on the left presents the PR-AUC scores. For example, the first column presents the PR-AUC when models trained on the KC1 dataset (horizontal axis) were evaluated against the others (vertical axis), with the first row blank to indicate the exclusion of the KC1 dataset in this phase. Similarly, the matrix on the right showcases the g-mean results under the same conditions.	43
3.1	The first column (left) of the nodes denotes developers with feature vectors a_{dq} , the middle column represents commits with feature vectors x_{cp} and the last column (on the right) represents the repositories to which commit has been made with feature vectors b_{rs}	53
3.2	Circles = commits; blue = non-defective, red = defective, Triangle-Up = lines of code in the file before the change (representative of the size of repository), Square = developer	55
3.3	Comparison of a simple GCN structure to an R-GCN structure. GCN model is more suitable for data structures involving a single relation type. However, R-GCNs are better suited for multi-relational type data.	57

Chapter 1

Graph-Based Feature Engineering for Just-in-Time Software Defect Prediction

1.1 Introduction

The growing prevalence of software defects is becoming a significant concern in the domain of software development [23]. These software defects, also known as bugs or errors, refer to flaws or mistakes in the design, implementation, or functionality of a software system. These errors often arise from mistakes committed by developers.

Software defects can emerge from a variety of sources throughout the development life of a software and recent advances in software development may have inadvertently contributed to their exacerbation. While the adoption of open-source software, facilitated by platforms like GitHub, has brought numerous benefits to software development, such as enhanced collaboration, knowledge sharing, and research opportunities [59], it has also introduced inherent risks. The open accessibility of source code and development history in open-source projects pose serious threats to the security of software systems, necessitating robust measures to ensure their protection. The complexity and interdependencies of modern software systems amplify the chances of committing a defect [52].

Software defects have been associated with a wide range of incidents that have exposed software systems to exploitable vulnerabilities, including incidents like Heartbleed [2] and Shellshock [1]. Additionally, a vulnerability in Apache Struts in 2017 led to the compromise of financial data for 143 million consumers [25]. A report published by the Consortium for Information and Software Quality sheds light on the severity of the issue. According to their findings, the total cost of poor software quality in the US reached an astonishing \$2.08 trillion in the year 2020 [40]. These pivotal incidents underscore the importance of investing efforts in improving the detection of potentially defective software changes at the earliest possible stage. Timely identification allows for prompt mitigation measures, reducing exposure and minimizing vulnerability.

Consequently, the increasing complexity of software defects has generated substantial attention in the field of just-in-time software defect prediction (JIT-SDP). JIT-SDP techniques aim to predict and identify software defects early in the development process, allowing for timely remediation by developers. Previous research has made significant strides in exploring various machine learning approaches, such as deep learning and data analysis techniques, for software defect prediction (SDP) [36, 48, 37, 38, 80]. These studies have showcased the potential of JIT-SDP models in improving the security of software systems by analyzing historical data that includes code changes and other relevant factors.

However, despite the recent achievements in predicting and identifying software defects, defects still go undetected [54] and hence, there is a timely need for further advancement. This can be made possible through leveraging the underlying structure of the software development process data. In the context of software development, three key entities play crucial roles: developers, commits, and repositories. Within the realm of metadata analysis, there are inherent interdependencies among these entities. Each commit represents a set of changes made by a developer, and these commits are often associated with specific repositories that serve as containers for storing and organizing the codebase. By incorporating these interdependencies into the prediction process, it is possible to enhance the precision and reliability of JIT-SDP models. This highlights the need for continued research and exploration in this area to further improve the prediction and prevention of software defects.

In this study, we propose a novel graph-based learning technique to formally account for these interdependencies and make improvements in defect detection performance. That is, by representing the relationships between developers, commits, and repositories as a graph structure, we aim to capture the inherent interdependencies that may not be readily apparent using traditional modeling techniques. Specifically, we introduce a graph-based defect ratio (GDR) feature that captures the number of defective commits associated with each commit. This feature provides valuable information about the historical defect patterns that allow us to quantify the potential risk associated with different entities in the software development process.

We evaluate our approach using six large-scale software project datasets from different communities namely Bugzilla [11], Columba [13], Eclipse JDT [12], Mozilla [11], Eclipse Platform [12], and PostgreSQL [15]. We employ the existing defective commit labels in these datasets to train binary supervised classification models that are specifically designed to detect and identify defective commits.

The main contributions of this study are as follows:

- We propose a unique graph-based feature, GDR, that encapsulates the interconnections among developers, commits, and repositories, that can not be captured by conventional defect prediction models. This feature serves as a cornerstone in understanding and formalizing the interdependencies inherent in the software development process.
- By integrating the GDR feature into our models, we facilitate the early detection of defective commits with a degree of explainability, empowering developers to proactively mitigate potential issues.
- Our comparative analysis indicates that the GDR exhibits promising potential as an alternative to traditional features in software defect prediction.

The remainder of the manuscript is organized as follows. In Section 1.2, we conduct a comprehensive review of the relevant literature concerning JIT-SDP and further highlight the current research gaps and our contributions. In Section 1.3, we provide an overview of the datasets used, present the problem formulation, describe the models employed, introduce

the evaluation metrics utilized, and describe the experiments conducted. The results for each of the experiments is presented in Section 1.4. Finally, Section 1.5 concludes the paper by summarizing the findings and outlining potential avenues for future research.

1.2 Background, Gaps, and Contributions

In this section, we first examine the related literature on JIT-SDP. Next, we highlight the current gaps and discuss our main contributions.

1.2.1 Just-in-Time Software Defect Prediction

Extensive research within SDP [77] has given rise to a sub-area known as JIT-SDP. Recently, several studies have leveraged metadata analysis to build early defect prediction models that effectively identify defective commits as soon as they are introduced into the system. Based on their labeling approach, these models can be divided into supervised and unsupervised categories. Supervised models use well-defined labels differentiating between defect-inducing or clean commits, while unsupervised models do not rely on labeled data. Additionally, the models can be classified according to their evaluation methodologies as within-project, which predicts defects within a single project, and cross-project, which leverages data from multiple projects to make predictions.

Majority of JIT-SDP models rely on supervised machine learning techniques. Among the prominent supervised models are random forests [36] and regression [48, 37]. In [36], the authors employed both within-project and cross-project testing on random forest models. The median area under the curve (AUC) values were approximately 78% and 71%, respectively. Further, in [38] the authors used support vector machines to classify software changes as similar to prior defective changes or clean changes. A recent systematic survey [77] suggested that ensemble learning techniques could achieve higher performance in the domain of JIT-SDP with F1 scores ranging from 25% to 75% [74]. Deep learning has gained popularity in this field [79, 34, 80]. For example, [80] proposed a deep learning model named DAECNN-JDP, that uses denoising autoencoder and convolutional neural network. Additionally, Graph-based learning techniques have become increasingly popular.

For instance, [16] employed an undirected bipartite graph to extract graph features using Neo4j. The results of the study exhibit remarkable potential by surpassing current JIT-SDP techniques with an AUC of approximately 82%. Despite their growing popularity, supervised models face a significant challenge due to limited labeled and/or imbalanced data.

Unsupervised models prove particularly valuable by addressing imbalanced data concerns without relying on labeled data. Previous research on defect prediction has revealed that the majority of defects (80%) tend to manifest in a concentrated subset (20%) of modules [14]. This distribution leads to an imbalanced dataset, where the predictive accuracy of the less frequent class instances is compromised due to the skewed class distribution. Anomaly detection is a popular technique that has been used in recent literature, to address the problem of imbalanced data. For instance, [44] explored the use of anomaly detection for fine-grained JIT defect prediction, achieving an AUC of 71%. Another study [76] proposed to transform the binary classification problem into an anomaly detection problem to effectively tackle the class imbalance problem and partially mitigate the issue of limited labeled data. This study also achieved an AUC of approximately 71%. Resampling the data is another popular technique to tackle the problem of imbalanced data [45, 69, 72]. It includes methods like undersampling the overrepresented class and/or oversampling the underrepresented class [56]. Despite the extensive research conducted on unsupervised models [43, 75], it is important to acknowledge that their predictive performance may not consistently reach the same competitive levels as their supervised counterparts [77].

Numerous contemporary studies used a standardized set of 14 widely recognized features, classified into five dimensions: Diffusion, Size, Purpose, History, and Experience [48]. These dimensions provide valuable insights into estimating the failure probability of code changes based on information available after the coding stage. Research conducted in studies like [36] and [37] incorporated these 14 factors in their investigations, using both open-source and commercial projects. These studies established benchmarks for the effectiveness of JIT-SDP techniques both for within-project predictions and cross-project predictions. In response to the growing importance of these dimensions, [43] proposed a metric that captured the total number of added and deleted lines. This new metric showed promising results

and outperformed some state-of-the-art techniques resulting in about 57% improvement in performance over the baseline results in [37].

Various other approaches have also been employed to tackle the challenges in JIT-SDP, such as studying the relationship between changes and bugs by examining version archives like Concurrent Versions System (CVS) and Bugzilla. For instance, [65] conducted an analysis of factors such as the authors of the change, the nature of the change, and the reasons behind the change, to identify properties of changes that may indicate potential problems. Furthermore, incorporating concept drifts and verification latency has demonstrated the potential to enhance SDP performance, providing up to 42% more stable results in terms of standard deviation and g-mean [17].

In summary, supervised learning techniques have exhibited the most promising outcomes so far. Furthermore, incorporating features that capture and/or simplify the intricate relationships among input data seems to be one of the ways in which past models have achieved improved performance. Therefore, these insights, combined with the need for explainability and actionable insights, could drive the next generation of JIT-SDP.

1.2.2 Gaps and Contributions

There is currently a need for improved and explainable JIT-SDP. In this study, we propose a novel, explainable *graph-based feature* and demonstrate that this feature elicits substantial information that may lead to improved model performance. Specifically, this feature captures the interconnections among entities within a two-step range of each commit. In the first step, it identifies how many connections each commit has. In the second step, it determines how many of the connections identified in step one are directly connected to a defective commit. This is done by identifying the two-hop neighbors of each commit. These neighbors can be linked via the same developer or repository. That is, any two commits that are made by the same developer and/or belong to the same repository are identified as two-hop neighbors. By considering these two steps, this novel feature allows us to capture a more comprehensive view of how a randomly selected commit is related to defective commits.

This proposed approach is different from past works. Perhaps, the most closely related work to ours was used in [16], which leverages *graph-based learning*. In this study, the authors

built contribution graphs consisting of developers and source code files to extract features from the one-mode projection graphs. Different from [16], however, our study introduces the novel graph-based feature to distinctly represent the relationship of each commit with other defective commits. This enables us to integrate our graph-based feature into standard classification models, preserving the versatility of the proposed approach.

1.3 Data and Methods

This section provides an overview of our data, the learning problem formulation, models, evaluation metrics, and experiments conducted.

1.3.1 Data

To enable comparisons with the literature, we used the six classical datasets previously examined in various studies (e.g., [37, 78]). By using the same datasets, we can establish a basis for comparing our results with the results of prior studies and assess the validity of our findings. The data in these six open-source projects, namely Bugzilla, Columba, Eclipse JDT, Mozilla, Eclipse Platform, and PostgreSQL, had been extracted from the CVS repositories associated with these projects and combined with relevant bug reports. For the Bugzilla and Mozilla projects, the data was provided by the MSR 2007 Mining Challenge [11], while for the Eclipse JDT and Platform projects, data was obtained from the MSR 2008 Mining Challenge [12]. In the case of Columba [13] and PostgreSQL [15], a replica of the official CVS repository was created. The summarized statistics for each dataset, including the duration, total number of changes, and the proportions of defective changes, are presented in Table 1.2. It is important to note that all six datasets exhibit some level of class imbalance. Among them, Mozilla stands out as the most imbalanced dataset, with only 5% of commits being defective. On the other hand, Bugzilla appears to be the most balanced dataset, with 36% of commits being classified as defective. The diverse class imbalance across the datasets allows for the further examination of the generalizability of the results. Each of these six datasets consists of 14 features categorized into five distinct dimensions, listed in Table 1.1. We have opted to use these features for consistency based on their successful track record in

traditional defect prediction research. However, we have taken these features a step further by incorporating them to create the new metric, GDR. We believe that this new metric adds value above and beyond the previously explored features.

1.3.2 Problem Formulation

Consider dataset D , which contains n software commits. Let commit c be represented by feature vector $X_c = [x_{c1}, x_{c2}, \dots, x_{cp}]$, $c = 1, 2, \dots, n$, where p is the number of features associated with each commit. Furthermore, let $Y = [y_1, y_2, \dots, y_n]$ denote the target vector, representing the status (i.e., labels) of the commits, where y_c equals 1 if commit c is defective and equals 0, otherwise. Our goal is to find the function $f(X)$ such that the loss function $L(f(X), Y)$, which quantifies the discrepancy between the predicted labels $f(X)$ and the true labels Y , is minimized.

To extract insights from the input data, we analyze it as a tripartite graph $G = (V, E)$, where V is the set of all vertices and E is the set of all edges (Fig. 1.1). The vertices in V are divided into three disjoint subsets: V^D , V^C , and V^R representing developers, commits, and repositories, respectively. We create the vertices representing developers and repositories using the feature vector X_c . This vector contains specific features: x_{cd} and x_{cr} . The feature x_{cd} corresponds to the ‘developer class’ for each commit, and x_{cr} corresponds to the ‘repository feature’ of each commit. Essentially we use these features to form two types of vertices: one that represents the developer involved (based on the developer class feature), and another that represents the repository associated with the commit (based on the repository feature). Each edge in the tripartite graph G can only connect nodes across different subsets [20]. An edge $(v_d, v_c) \in E$ connects a developer vertex, $v_d \in V^D$, to a commit vertex, $v_c \in V^C$, if the commit was made by the developer belonging to that class. An edge $(v_c, v_r) \in E$ connects a commit vertex, $v_c \in V^C$, to a repository vertex, $v_r \in V^R$, if the commit is made to a file represented by the corresponding repository feature. And, an edge $(v_d, v_r) \in E$ connects a developer vertex, $v_d \in V^D$, to a repository vertex, $v_r \in V^R$, if a developer belonging to that class commits to a file represented by the corresponding repository feature.

Table 1.1: Traditional Features Characterized by 5 Dimensions

	Name	Definition	Rationale
Diffusion	NS	Number of modified subsystems	Changes modifying many subsystems exhibit higher chance of a defect
	ND	Number of modified directions	Changes modifying many directions exhibit higher chance of a defect
	NF	Number of modified files	Changes modifying many files exhibit higher chance of a defect
	Entropy	Distribution of modified code across each line	Higher entropy exhibit higher chance of a defect
Size	LA	Lines of code added	The more lines of code added, the higher the chance of a defect
	LD	Lines of code deleted	The more lines of code deleted, the higher the chance of a defect
	LT	Lines of code in a file before the change	The larger a file, the more likely a change may be defective
Purpose	Fix	Whether a change is a defect fix	Area of fixing a defect may be more defect prone
History	NDEV	Number of developers that changed modified files	Large the number of developers making the change, higher the chance of a defect
	AGE	Average time between last and current change	Lower the age, higher the chance of a defect
	NUC	Number of unique changes to the modified files	Larger the NUC, higher the chance of a defect
Experience	EXP	Developer experience	Experience of a developer will be inversely proportional to the chance of defect
	REXP	Recent developer experience	Developer who made modifications more recently is less likely to introduce a defect
	SEXP	Developer experience on a subsystem	Developer more familiar with the subsystems is less likely to introduce a defect

Table 1.2: Statistics of the datasets used in our study

	Time	No. of Changes	% Defective
BUG	1998.08-2006.12	4,620	36%
COL	2002.11-2006.07	4,455	31%
JDT	2001.05-2007.12	35,386	14%
MOZ	2000.01-2006.12	98,275	5%
PLA	2001.05-2007.12	64,250	14%
POST	1996.07-2010.05	20,431	25%

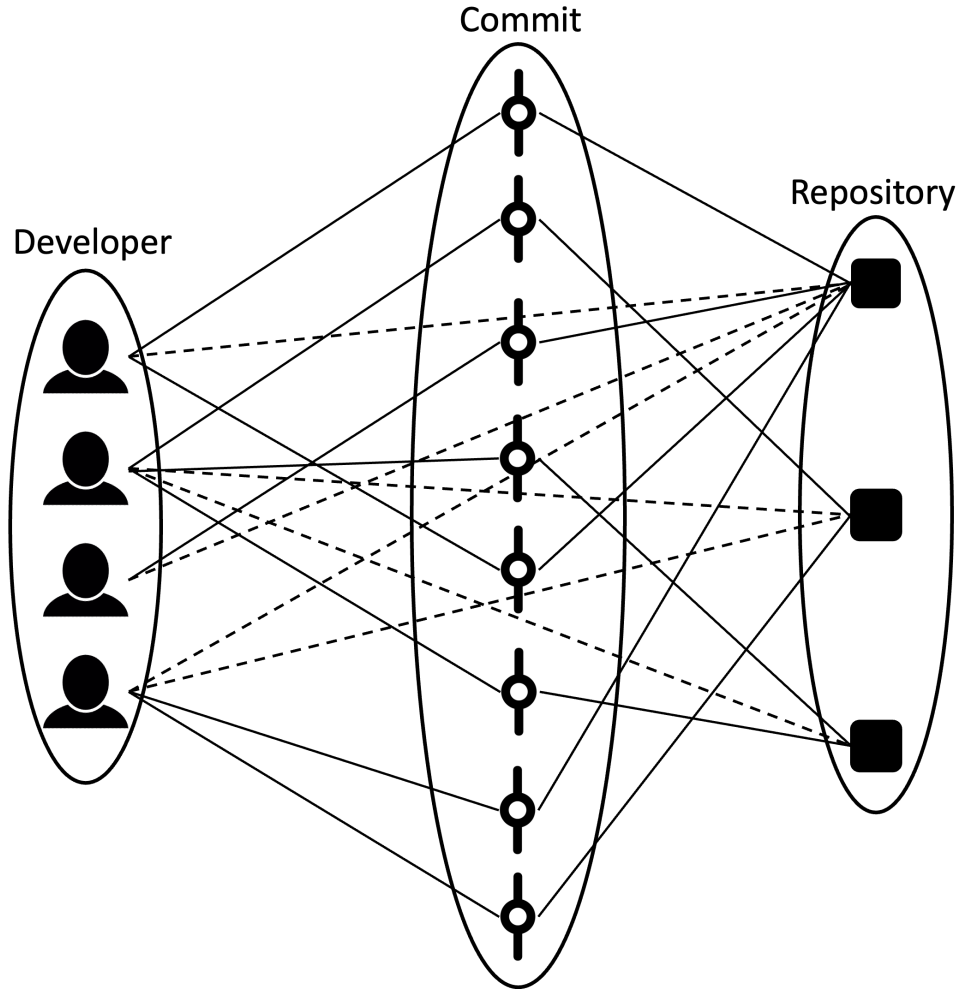


Figure 1.1: The nodes on the left, that is $v_d \in V^D$, represent the developer's label, nodes in the middle column, $v_c \in V^C$, represent the commits, and nodes in the right column, $v_r \in V^R$, represent the repository to which the commit has been made.

We now delve into the construction of the three components of the tripartite graph using Fig 1:

- **Developer:** The first column on the left represents developers, with each node corresponding to a unique developer class. We determine these classes based on the developer’s level of experience, with each group covering a range of 10 units. For example, developers with experience ranging from 1-10 are grouped together under one class, while those with experience ranging from 10-20 are grouped under another class, and so on. For clarity, each node here corresponds to a unique developer class, which is determined using the feature value x_{cd} from the feature vector X_c . Essentially, these nodes categorize commits based on the class of the developer involved in each commit.
- **Commit:** In the center column, every node corresponds to a single commit.
- **Repository:** The last column represents the repository entity. Here each node corresponds to a unique repository feature x_{cr} , as defined by the feature value in X_c . Since all the commits within any file belong to the same repository, we do not have a dedicated category for this entity. Instead, we rely on the ‘lines of code in the file before the change (lt)’ feature to signify this entity, as it is our most effective means of representing a repository.

It is important to note that the tripartite graph is created solely to generate GDR for each commit. Once calculated, we focus on a comprehensive set of experiments that examine the effectiveness and relevance of GDR as a standalone feature or in comparison to the 14 standard features listed in Table 1.1. These experiments are designed to rigorously evaluate the predictive power and utility of GDR in the context of software defect prediction. To facilitate this, GDR is computed using two specific metrics:

- **2-hop degree of a commit ($d_2(v_c)$):** For each commit node, v_c , we calculate its degree with 2-hop neighborhood. That is, we determine the total number of nodes that are directly connected to that commit node and the nodes that are connected to those directly connected nodes.

- 2-hop defective node count of a commit ($d_{2_d}(v_c)$): For each commit node, v_c , we determine the total number of defective commit nodes within the same 2-hop neighborhood.

The GDR for each commit is calculated using the following formula:

$$\text{GDR} = \frac{d_{2_d}(v_c)}{d_2(v_c)}.$$

Note that GDR is particularly valuable in identifying potentially defective commits. That is, a high GDR for a node suggests that the commit is part of a larger network of defective activities, while a low GDR indicates that it is less likely to be defective.

1.3.3 Models

We applied the best-performing and widely used random forest classification model to analyze our datasets. The random forest is an ensemble learning technique that aggregates many decision trees to make predictions [33]. To address class imbalance, we implement a balanced random forest model that assigns weights to each observation based on class frequency. This ensures that both the minority (defective commits) and majority (non-defective commits) classes are given equal importance. We specifically select random forest for this study as this algorithm can handle noisy data, is generally resilient to outliers, and is widely used in the literature [58, 77, 36, 74].

1.3.4 Metrics and Model Evaluation

We used several commonly used metrics to evaluate and compare our models, including accuracy, F1 score, precision, recall, and AUC. Accuracy measures the ratio of correct predictions to total predictions made, while F1 score combines precision and recall. Precision represents the proportion of accurately predicted positive instances out of all predicted positives and recall represents the proportion of accurately predicted positives out of all actual positives. AUC is a measure that reflects the overall ranking performance of a classifier, indicating its ability to differentiate between positive and negative instances [35].

Additionally, to ensure that the results capture any issues with respect to class imbalance which is prevalent in our datasets, we employ precision-recall AUC (PR-AUC). PR-AUC is especially useful when assessing binary classification models with imbalanced datasets, as it focuses on precision and recall to accurately identify the positive class [27, 66].

Our study leverages the intrinsic feature importance ranking capability of Random Forest classifiers to shed light on the significance of each feature within our models. This method ranks features based on how much they contribute to the improvement of the model’s predictive accuracy, providing a straightforward and interpretable measure of each feature’s relevance to the model’s decision-making process. We applied this feature importance analysis across the entire dataset to ensure a thorough evaluation. The outcomes of this analysis were critical in guiding our experimental design, particularly in selecting and prioritizing features for in-depth examination and comparison. This approach not only simplifies the interpretability of the model but also aids in identifying key features that have the most substantial impact on the model’s predictions.

To assess the efficacy of our random forest models across all datasets, we utilize 10-fold cross-validation [32] and report the mean and standard deviation of each metric. In each fold, training is done using balanced sets, i.e., equal numbers of defective and non-defective commits. We leveraged the scikit-learn library [55] to implement both our models. For the random forest model, we selected 100 estimators and for logistic regression, we used a maximum iteration of 10,000 for convergence.

1.3.5 Experiments and Rationale

This section discusses the experiments we conducted to evaluate the effectiveness of the GDR in our software defect prediction framework. Each experiment has been carefully designed to uncover the predictive capabilities of GDR and its ability to perform independently compared to traditional feature sets. Further, to prevent label leakage into the testing dataset, GDR was derived using features from both training and testing datasets, but only labels from the training set were employed in each of the experiments.

Features Benchmarking

In our preliminary experiment, we incorporated GDR with existing traditional features to configure a standard random forest model. This setup aimed to utilize the model’s inherent feature importance ranking capability to assess and prioritize the relevance of all included features, GDR among them.

Analyzing GDR’s Constituents

We carried out an analysis to compare the performance of a model using only GDR with a model based on the two traditional features, developer experience, and lines of code before the change, from which GDR is derived. This comparison aimed to assess whether GDR’s method of capturing interdependencies among developers, commits, and repositories offers an advantage over using the traditional features alone. Essentially, this experiment sought to demonstrate how GDR, by integrating more than just the basic features, might provide a more effective tool for prediction by incorporating the complex relationships within the software development process.

Performance Benchmarking

Next, we compared the performance of models using solely GDR as an input feature with the performance of the models that rely on the best-performing traditional feature(s). We used random forest models to analyze the performance across each of the six datasets individually. This experiment helped us assess the intrinsic value of GDR as an independent predictor and its potential to outperform the traditional features that have long dominated the landscape of software defect prediction.

Through these carefully designed experiments, we seek to thoroughly understand GDR’s impact and its promising role in JIT-SDP. Each experiment contributes to a clearer insight into GDR’s effectiveness and its importance in improving defect prediction models.

1.4 Results

In this section, we delve into the outcomes of the comprehensive experiments conducted across six datasets to assess the efficacy of the GDR. Our analyses delve into GDR’s intrinsic value, its comparative performance against traditional features, and its unique predictive capabilities, laying the groundwork for a detailed discussion on its role in enhancing defect prediction.

1.4.1 Feature Importance

In our study, we assessed how incorporating GDR alongside traditional features within a random forest model influences performance across six datasets. The results consistently showed GDR as the top-ranked feature in importance. While we specifically illustrate this with the feature importance plot for the Platform dataset (Fig 1.2), the pattern held true across all datasets examined. This underscores GDR’s significant contribution to enhancing the model’s defect prediction accuracy, suggesting that GDR effectively encapsulates the complex interplay of factors involved in software defects better than traditional features.

1.4.2 GDR Composition

We then compared the performance of models solely utilizing GDR with those based on its core elements: developer experience and lines of code in the file before the change. The comparison revealed that models powered by GDR outperformed those relying on these traditional features, underscoring GDR’s capability to synthesize complex interactions between developers, commits, and repositories. This advantage was particularly noticeable in recall and F1 scores within the JDT, Mozilla, Platform, and PostgreSQL datasets, as shown in Table 1.3. This table outlines the average and standard deviations for various metrics, demonstrating the comparative efficacy of the two model types across the datasets.

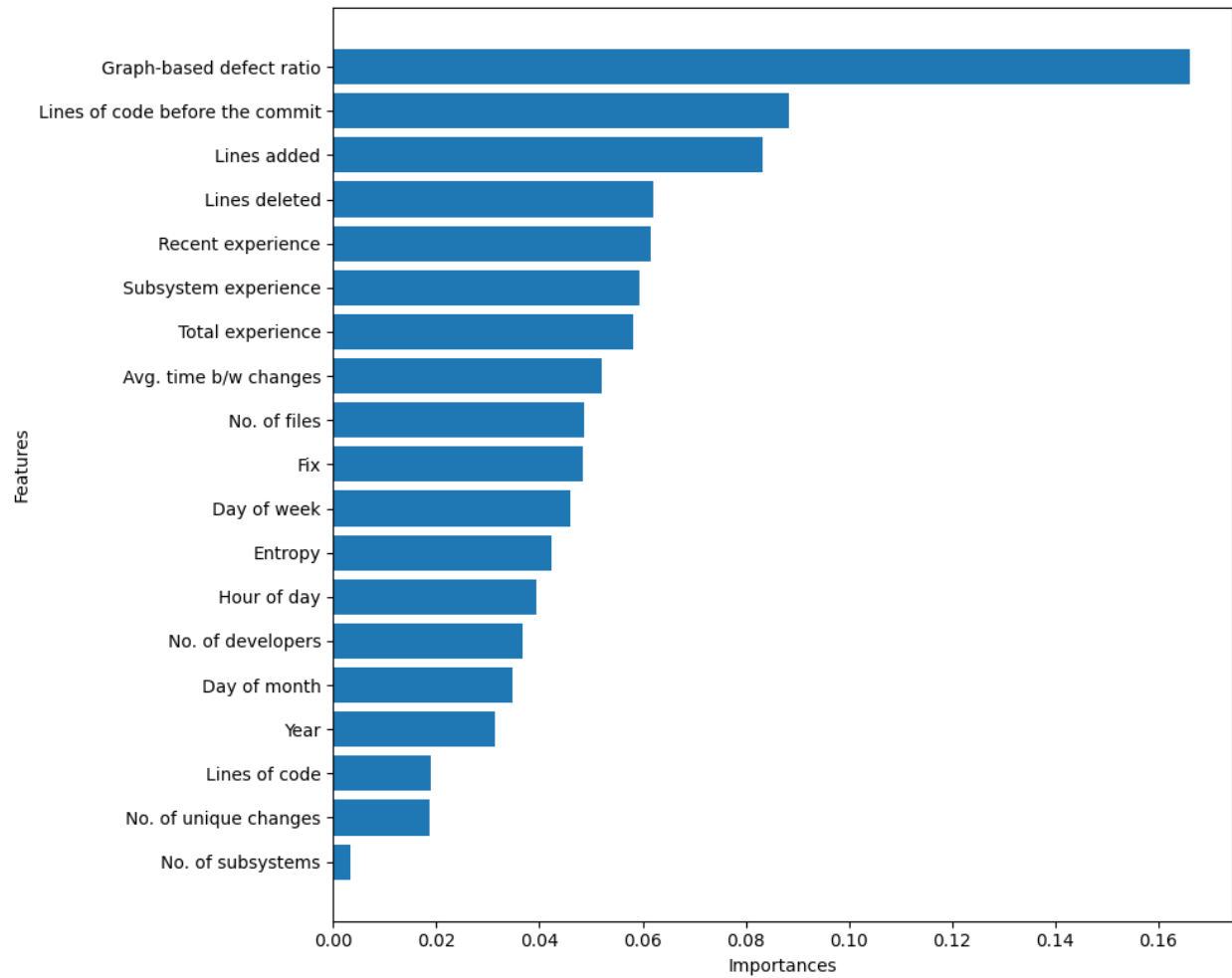


Figure 1.2: This plot offers feature importances based on their influence on the model's predictions. GDR has the highest importance followed by other features.

Table 1.3: Comparison of Model Performances Utilizing GDR, Developer Experience, and Lines of Code as Input Features

	Bugzilla		Columba		JDT		Mozilla		Platform		PostgreSQL	
	GDR	exp & lt	GDR	exp & lt	GDR	exp & lt	GDR	exp & lt	GDR	exp & lt	GDR	exp & lt
Accuracy	56.43%	58.20%	63.55%	67.95%	67.33%	83.50%	80.47%	94.30%	65.92%	82.96%	65.47%	70.90%
	$\pm 2.09\%$	$\pm 1.31\%$	$\pm 1.83\%$	$\pm 1.98\%$	$\pm 1.29\%$	$\pm 0.65\%$	$\pm 0.42\%$	$\pm 0.20\%$	$\pm 0.30\%$	$\pm 0.36\%$	$\pm 0.93\%$	$\pm 0.89\%$
Precision	35.68%	40.46%	35.42%	46.13%	15.72%	18.92%	5.42%	7.14%	17.48%	23.02%	27.25%	33.82%
	$\pm 3.66\%$	$\pm 4.13\%$	$\pm 4.88\%$	$\pm 2.97\%$	$\pm 1.72\%$	$\pm 3.15\%$	$\pm 0.57\%$	$\pm 3.54\%$	$\pm 0.82\%$	$\pm 2.60\%$	$\pm 1.64\%$	$\pm 3.51\%$
Recall	23.63%	30.02%	23.56%	29.55%	29.01%	4.52%	16.61%	0.74%	35.43%	6.69%	22.78%	16.75%
	$\pm 4.08\%$	$\pm 3.86\%$	$\pm 3.81\%$	$\pm 2.74\%$	$\pm 1.93\%$	$\pm 0.92\%$	$\pm 2.16\%$	$\pm 0.35\%$	$\pm 2.18\%$	$\pm 0.62\%$	$\pm 2.49\%$	$\pm 1.25\%$
F1	28.34%	34.42%	28.20%	35.97%	20.37%	7.29%	8.16%	1.34%	23.41%	10.37%	24.79%	22.39%
	$\pm 3.80\%$	$\pm 3.86\%$	$\pm 3.95\%$	$\pm 2.66\%$	$\pm 1.87\%$	$\pm 1.42\%$	$\pm 0.88\%$	$\pm 0.64\%$	$\pm 1.20\%$	$\pm 0.98\%$	$\pm 2.10\%$	$\pm 1.84\%$
PR-AUC	36.93%	39.74%	34.06%	44.00%	15.44%	17.33%	5.52%	6.66%	17.68%	19.54%	27.11%	31.25%
	$\pm 2.38\%$	$\pm 3.75\%$	$\pm 3.43\%$	$\pm 2.59\%$	$\pm 0.98\%$	$\pm 1.42\%$	$\pm 0.25\%$	$\pm 0.48\%$	$\pm 0.60\%$	$\pm 0.75\%$	$\pm 1.45\%$	$\pm 1.62\%$

1.4.3 Comparison Against Top Traditional Feature

For each of the six datasets, we meticulously compared the performance of a model solely utilizing GDR against another model employing the most effective traditional feature. This side-by-side evaluation brought to light critical insights. Table 1.4 outlines the average and standard deviations for various metrics, demonstrating the comparative efficacy of the two model types across the datasets.

- Accuracy: In the Columba dataset, the model using GDR as the sole input showcased higher accuracy than its counterpart using the best traditional feature, suggesting it more accurately mirrors the actual defect scenarios within these projects.
- Recall: For the JDT, Mozilla, and Platform datasets, the GDR-based models demonstrated higher recall, indicating GDR’s proficiency in identifying a wider range of true defects.
- F1 Score: For JDT, the GDR-based model exhibited higher F1 scores, indicating an optimal blend of precision and recall.
- Precision and PR-AUC: Across most datasets, the performance in precision and PR-AUC between the two models was comparable, with slight variations favoring neither consistently.

1.4.4 Comparative Evaluation with Composite Feature Sets

In this part of the analysis, we further explore the performance of the GDR-based model in comparison to models that employ a composite of top traditional features. The findings from the JDT dataset reveal that even the combined strength of the top three traditional features does not match the performance metrics achieved by GDR alone, showcasing GDR’s superior capability in accurately predicting defects. Similarly, in the Mozilla dataset, the inclusion of the top two traditional features was necessary to approximate the results obtained with GDR, reinforcing the potential of GDR as an effective alternative to traditional features. This experiment underscores the promise of GDR as a viable substitute for traditional feature sets in defect prediction models.

Table 1.4: Comparison of Model Performances Utilizing GDR vs. Top-Performing Traditional Feature

	Bugzilla		Columba		JDT		Mozilla		Platform		PostgreSQL	
	GDR	Top Feat.	GDR	Top Feat.	GDR	Top Feat.	GDR	Top Feat.	GDR	Top Feat.	GDR	Top Feat.
Accuracy	56.43%	58.25%	63.55%	60.49%	67.33%	76.46%	80.47%	89.32%	65.92%	67.86%	65.47%	65.64%
	$\pm 2.09\%$	$\pm 4.36\%$	$\pm 1.83\%$	$\pm 2.45\%$	$\pm 1.29\%$	$\pm 0.69\%$	$\pm 0.42\%$	$\pm 0.49\%$	$\pm 0.30\%$	$\pm 0.61\%$	$\pm 0.93\%$	$\pm 0.92\%$
Precision	35.68%	42.73%	35.42%	34.99%	15.72%	18.08%	5.42%	7.50%	17.48%	18.20%	27.25%	33.18%
	$\pm 3.66\%$	$\pm 5.61\%$	$\pm 4.88\%$	$\pm 3.37\%$	$\pm 1.72\%$	$\pm 1.53\%$	$\pm 0.57\%$	$\pm 1.14\%$	$\pm 0.82\%$	$\pm 0.65\%$	$\pm 1.64\%$	$\pm 1.22\%$
Recall	23.63%	39.55%	23.56%	34.51%	29.01%	17.99%	16.61%	9.12%	35.43%	33.88%	22.78%	36.67%
	$\pm 4.08\%$	$\pm 5.77\%$	$\pm 3.81\%$	$\pm 4.47\%$	$\pm 1.93\%$	$\pm 1.12\%$	$\pm 2.16\%$	$\pm 1.31\%$	$\pm 2.18\%$	$\pm 1.15\%$	$\pm 2.49\%$	$\pm 1.77\%$
F1	28.34%	40.96%	28.20%	34.71%	20.37%	18.01%	8.16%	8.22%	23.41%	23.67%	24.79%	34.83%
	$\pm 3.80\%$	$\pm 5.29\%$	$\pm 3.95\%$	$\pm 3.79\%$	$\pm 1.87\%$	$\pm 1.19\%$	$\pm 0.88\%$	$\pm 1.19\%$	$\pm 1.20\%$	$\pm 0.79\%$	$\pm 2.10\%$	$\pm 1.34\%$
PR-AUC	36.93%	42.92%	34.06%	35.80%	15.44%	16.19%	5.52%	5.81%	17.68%	19.21%	27.11%	31.54%
	$\pm 2.38\%$	$\pm 3.56\%$	$\pm 3.43\%$	$\pm 2.24\%$	$\pm 0.98\%$	$\pm 1.12\%$	$\pm 0.25\%$	$\pm 0.43\%$	$\pm 0.60\%$	$\pm 0.82\%$	$\pm 1.45\%$	$\pm 1.47\%$

1.5 Conclusion and Future Work

In our study, we put forth the GDR as a novel metric for predicting software defects, utilizing a random forest framework across diverse datasets. Our investigation into GDR’s inherent predictive ability showcased its potential, often outperforming traditional features it was derived from, especially in recall and F1 scores. This underscores GDR’s viability as an independent predictor in defect prediction, presenting a proof-of-concept for graph-based feature engineering in this field.

Our analysis extended beyond evaluating GDR in isolation, comparing its performance with models built on top traditional features. The results revealed that for multiple datasets, models built with only GDR consistently outperformed or matched models that relied on top-performing traditional feature. Specifically, models with GDR showcased superior recall and F1 scores, outshining the best individual traditional feature in the JDT, Mozilla, and Platform datasets. When we broadened the comparison to include models with multiple top-performing traditional features, the results were telling. Specifically, in the JDT dataset, replicating GDR’s effectiveness in terms of recall and F1 score required merging the top three traditional features. For Mozilla, the top two traditional features were needed to surpass the GDR model’s results.

It’s pertinent to highlight that while our discourse predominantly revolves around ”defective” commits, the versatility of GDR allows for a broader application spectrum. The binary classification system employed in our study—categorizing commits as ”defective” or ”non-defective”—can be recalibrated to suit varied analytical needs, extending the utility of GDR to other domains such as vulnerability detection or code quality assessment.

Our study highlights the GDR as a potent alternative to a broad spectrum of traditional features in software defect prediction, underscoring the need for further exploration to fully leverage its advantages in enhancing modeling efficiency in software engineering. The efficacy of GDR as an independent predictor, capable of outperforming or equating the predictive power of traditional features, advocates for its potential as a formidable substitute in defect prediction models. This is particularly compelling given the complexity and resource demands associated with managing and analyzing a vast array of traditional features. The

adoption of GDR could streamline the modeling process, offering a more simplified yet equally effective approach by encapsulating the intricate interdependencies within software development into a single, comprehensive metric.

As a part of future work, we recommend thoroughly exploring the capabilities of GDR within advanced graph-based analytical frameworks to enhance the effectiveness of defect prediction models. And to delve into the applications of graph neural networks (GNNs) to extract more sophisticated features that GDR could encompass, thereby enriching the models. Furthermore, examining various graph representations can help understand their influence on the effectiveness of GDR and overall model performance across different metrics. Further exploring the utility of GDR and related graph-based methods is expected to contribute to valuable insights into the field, potentially leading to meaningful improvements in predictive modeling techniques.

Chapter 2

Generative Adversarial Network Based Anomaly Detection for Software Defect Prediction

2.1 Introduction

Software defects, commonly referred to as bugs or errors, continue to present challenges in the field of software development. These defects often originate from developer oversights or various factors throughout the software development process. While open-source software offers numerous advantages, its publicly accessible source code can sometimes expose vulnerabilities, escalating security concerns. Further, the growing complexity of modern software systems increases the likelihood of defects. Such defects have been linked to high-profile incidents like Heartbleed [2], Shellshock [1], and Apache Struts [25] emphasizing the need for early defect detection.

Supervised learning techniques, notably random forests [48, 36] and regression methods [37], have been extensively employed in software defect prediction, showcasing their promise. However, they come with a significant caveat – they rely on abundant, well-annotated datasets, a resource that may not always be readily available in practice. Furthermore, it is a common observation that a significant proportion of defects in the domain of software defect

prediction, often surpassing 80%, are concentrated in a relatively small portion of the dataset, typically 20% or even less [67]. This skewed distribution introduces a challenge known as class imbalance when constructing classifiers. These classifiers tend to prioritize the majority class, which constitutes most of the instances, potentially neglecting the minority class. Consequently, this imbalance issue leads to a situation where the predictive performance for the minority class lags behind that of the majority class [72]. Additionally, software defects can often be subtle anomalies, concealed within the code and only becoming apparent when they cause issues. These software anomalies, representing uncommon or defective patterns, are naturally rare, making them hard to detect using conventional supervised methods. Given this elusive nature, there’s a growing emphasis on employing anomaly detection techniques. Such methods are crucial as they offer a robust approach to identify and address these intermittent software defects before they escalate.

The domain of anomaly detection has witnessed multiple studies exploring its utility in diverse applications [5, 4, 50, 19]. As researchers have delved deeper into this area, a new kind of model has gained attention: Generative Adversarial Networks or GANs [30]. These networks are becoming increasingly popular because they can work with limited labeled data. Their ability to pick out minor differences from what’s considered ‘normal’ makes them especially useful for spotting anomalies across different areas.

Consequently, we apply GAN-based anomaly detection (GAN-AN), a modified version of the standard GANs for detecting anomalies in software defects. This approach consists of two encoders, a decoder, and a discriminator. These elements work together to classify a data point as either normal or anomalous [6]. Additionally, we refine three well-known anomaly detection models: autoencoders [29], isolation forest [42], and the one-class support vector machine (OCSVM) [62]. By comparing the results from GAN-AN to these models, we aim to highlight the potential benefits of expanding the application of our proposed method in efficiently spotting anomalies. To achieve this, our model is exclusively trained on data characterized as normal, which, in this context, denotes defect-free commits. During the testing phase, the model’s core objective is to identify the instances that deviate from the established norm of defect-free commits, highlighting the anomalies. This method is not just a theoretical exercise; it mirrors the tangible challenges software testers face in real-world

scenarios. In many situations, the available data on defects is sparse, making traditional identification methods less effective. Moreover, the nature and characteristics of defects can shift due to a range of factors, varying from changing development approaches to the introduction of new technologies.

In this research, we employ five distinct open-source NASA datasets to evaluate our proposed method. Each of these datasets have labels indicating whether the commit is defective, providing a ground truth for our investigations. Our objective is to devise models proficient in detecting these defects. To ensure a thorough evaluation of our models, we adopt a two-fold testing approach including: 1) *Within-Project Testing*: In this strategy, both the training and testing phases are conducted using data sourced from the same project. This approach offers insights into how well the model recognizes and adapts to the nuances and patterns inherent to that specific project. And, 2) *Cross-Project Testing*: To test the generalizability of our models, we use a cross-project approach where the model is trained using data from a single project, and its performance is subsequently assessed on the remaining four projects. This method tests the model’s capability to generalize its understanding and apply it to unfamiliar datasets. By adopting this dual evaluation method, we gauge both the depth (specific understanding) and breadth (general applicability) of our model’s proficiency.

The main contributions of this study are as follows:

- We apply a GAN-based anomaly detection technique to identify potentially defective commits.
- We develop three additional anomaly detection models namely autoencoders, isolation forest, and OCSVM, and compare the results of the four models.
- We compare the efficiency of our models against state-of-the-art models and demonstrate their superiority.
- Finally, we evaluate the success of our approach through cross-project testing, showing its generalizability.

The remainder of the manuscript is organized as follows. In Section 2.2, we conduct a comprehensive review of the relevant literature concerning software defect prediction and further highlight the current research gaps and our contributions. In Section 2.3, we provide an overview of the datasets used, describe the models employed, and introduce the evaluation metrics utilized. In Section 2.4, we present both within-project testing and cross-project testing results. Finally, Section 2.5 concludes the paper by summarizing the findings and outlining potential avenues for future research.

2.2 Related Work, Gaps and Contributions

In this section, we first examine the related literature on using unsupervised and semi-supervised techniques for Just-in-time software defect prediction (JIT-SDP). Next, we highlight the current gaps that necessitate the need for anomaly detection.

2.2.1 Background

Recent research has delved into various methods to enhance SDP strategies, ranging from the incorporation of unsupervised and semi-supervised learning to the use of anomaly detection mechanisms. Unsupervised learning operates with data that lack specific labels. Conversely, semi-supervised learning utilizes a combination of labeled and unlabeled data, drawing insights from the substantial pool of unlabeled data. Anomaly detection, however, zeroes in on detecting uncommon occurrences, such as defects in commits. In this section, we first delve into works centered on semi-supervised and unsupervised methodologies, followed by a discussion on research specifically employing anomaly detection techniques.

In recent literature, there has been a growing interest in leveraging unsupervised and semi-supervised learning techniques for defect prediction. The use of self-supervised training with large unlabeled datasets to reduce reliance on labeled data and improve model understanding was highlighted by [7]. The study also noted the lack of standardized benchmarks and metrics for model evaluation, which makes it challenging to compare different approaches. Consequently, [49] explored unsupervised models as an alternative to supervised methods, focusing on outlier detection. Their approach was inspired by the Pareto principle, which

states that a small number of modules contain the majority of defects. Experimental results from 16 software projects, utilizing publicly available datasets (PROMISE and GitHub), show that in the best-case scenario, the k-Nearest Neighbor outlier detection method effectively identifies 94% of expected defects, and in 75% of the projects, it identifies over 63% of the defects. Further, to tackle the challenge of rare faulty samples for training, [21] proposed a one-class software fault prediction model using only non-faulty samples based on OCSVM. The study used six extremely imbalanced datasets from the real world. The new technique showed promise with an average geometric mean (g-mean) increase over all datasets to 68.8% from 59.5% using previous techniques.

The escalating interest in anomaly detection underscores its significance, especially in the context of imbalanced data. Recently, [76] proposed ADGAN-SDP, an anomaly detection model that effectively handles class imbalance and limited labeled data by transforming the binary classification problem into an anomaly detection problem. The ADGAN-SDP technique outperforms eight baseline models on NASA [63], AEEEM [26], and ReLink [73] repositories in terms of recall. Further, [3] introduced the Reconstruction Error Probability Distribution (REPD) model that incorporates autoencoder network for anomaly detection. REPD leverages the idea that defective code exhibits unusual patterns or structures compared to non-defective commits. REPD demonstrated promising results with an improvement in the F1-score up to 7.12%. Other works, like [44], also delve into anomaly detection’s role in defect prediction, shedding light on its potential and challenges. Additionally, [24] introduced the isolation forest technique to address challenges posed by unbalanced datasets. Their research demonstrated the effectiveness of this method, particularly when carefully selecting features as opposed to random selection. Another study was conducted in 2017 [51] using NASA datasets collected from PROMISE repository [60] to model software defects anomalies using univariate and multivariate Gaussian distribution models, considering both computational efficiency and correlation between attributes. Experimental results demonstrate that anomaly detection techniques perform as well as state-of-the-art classifiers without data preprocessing. The proposed anomaly detection approach aims to tackle the class imbalance problem and shows promising results compared to traditional methods.

Various other approaches have been employed to tackle two major challenges in software defect prediction namely noise and class imbalance. For eg., [28] introduced cluster-based over-sampling with noise filtering (KMFOS). This approach divides defective instances into clusters and generates new defective instances through interpolation between clusters, mitigating class imbalance and improving predictive model performance in SDP. This paper explores KMFOS in detail and presents empirical evidence of its effectiveness. Further, [53] utilizes a large dataset consisting of open-source software projects to address the noise and class imbalance challenge. The authors manually added noise to these datasets to identify its impact on the performance of those SDP models. The findings indicate that the introduction of noise led to a notable improvement, with a reduction in error rates of 40-60% when compared to traditional models.

In summary, there has been considerable work in applying semi-supervised techniques, like anomaly detection, to tackle the challenges of software defect prediction. These methods are proving to be just as effective as supervised approaches in some cases. Given the challenges with data availability and quality in JIT-SDP, these semi-supervised methods could be crucial for future advancements in this field.

2.2.2 Gaps and Contributions

Recent literature underscores the increasing importance of Unsupervised Software Defect Prediction (UnSDP) techniques. A comprehensive survey by [41] reviewed 49 studies and underscored that UnSDP methods, which do not rely on labeled data, often yield performance at par with their supervised counterparts. Further investigation into elementary unsupervised models for effort-aware JIT defect prediction was conducted by [75], revealing promising outcomes. Next, [8] demonstrated that while supervised models excel with ample labeled data, unsupervised methods become essential in scenarios where data is either unlabeled or imbalanced which is a common challenge in the field of predicting software defects. Indeed, unsupervised models, particularly clustering-based approaches, are notably advantageous when the availability of labeled data is scarce. Therefore, it becomes imperative to thoroughly investigate the intricate relationship between the characteristics of

datasets and the performance of prediction models, as well as to comprehend the dynamics of how datasets influence learning algorithms.

Although there have been advancements in semi-supervised methods, such as those applied to anomaly detection, the field remains ripe for further exploration. To address this, we use GAN-AN an innovative technique built upon the principles of GANs. GAN-AN is engineered to leverage the distinctive capabilities of GANs for the effective identification of anomalies. Our approach seeks to provide a fresh perspective and potential improvements in the field of semi-supervised learning techniques.

2.3 Methods

2.3.1 Data

To facilitate effective cross-referencing with earlier research, we have incorporated five publicly available NASA datasets from the Promise Software Engineering Repository [60]. These datasets were meticulously curated by mining modules from NASA products. Specifically, we have utilized the CM1, JM1, KC1, KC2, and PC1 datasets. Each of these datasets constitutes 21 distinctive features derived from the Halstead [31] and McCabe [47] metrics. A comprehensive overview of these datasets can be found in Table 2.1. Furthermore, in Table 2.2, we present a detailed account of these features along with their descriptions.

2.3.2 Models

In our study, to ensure a thorough analysis, we compare our GAN-AN model with three well-known anomaly detection techniques: autoencoders, isolation forest, and OCSVM. Initially, we delve into the workings of standard autoencoders to establish a baseline for anomaly detection. We then transition to a detailed discussion on GAN-AN, showcasing how it builds upon and surpasses the capabilities of both GANs and traditional autoencoders. This sets the stage for further comparisons with the isolation forest and OCSVM methods, allowing us to investigate the performance of our proposed model within the broader spectrum of anomaly detection techniques.

Table 2.1: Statistics of the datasets used in our study

Dataset	Language	Dataset Size	Defective Count	% Defective
KC1	C++	2,109	326	15.46%
KC2	C++	522	107	20.50%
CM1	C	498	49	9.84%
PC1	C	1,109	77	6.94%
JM1	C	13,204	2,103	15.93%

Table 2.2: Features Descriptions

Features	Description
loc	Module total number of line count
$v(g)$	Cyclomatic complexity measurement (McCabe)
$ev(g)$	McCabe complexity
$iv(g)$	(McCabe) complexity design analysis
n	Module numeral operators
v	Volume of each module
l	Length of program
d	Difficulty measurement
i	Intelligence measurement
e	Error approximation
b	Effort's estimation
t	Estimator of time
lOCode	Software module line of code
lOComment	Software module line of comment
lOBlank	Number of total blank lines in the module
locCodeAndComment	Number of the total line of code and comments
uniq_Op	Number of total unique operators
uniq_Opnd	Number of total unique operands
total_Op	Number of total operators
total_Opnd	Number of total operands
branchCount	Number of total branches in the software module
defects	Whether defect is present or not

Autoencoder

Autoencoders are a specialized type of neural networks designed to identify anomalous patterns within sequential data. They function by compressing high-dimensional data into a more compact, lower-dimensional representation through an encoder. This condensed essence of the input is then expanded back to its original dimensionality by a decoder. The primary objective of an autoencoder is to achieve a reconstructed output that is as close to the input as possible. The difference between the input and the reconstructed output is quantified using a loss function, often based on measures like mean squared error. When a data point's reconstruction loss exceeds a certain threshold it is considered as an anomaly. This threshold is typically determined through validation on a non-anomalous dataset which is considered 'normal'. These anomalies are indicative of patterns that do not conform to the model's learned representation of normalcy.

Generative Adversarial Network Anomaly Detection

The GAN-AN [6] model, inspired by the architecture of GANs [30], serves as a robust anomaly detection system. As illustrated in Fig. 2.1, the model involves two encoders, a decoder, and a discriminator, each playing a vital role in identifying varied data patterns. The GAN-AN operational process begins by compressing complex input data into a manageable, encoded representation using the first encoder. This encoded form is given to the decoder, which aims to accurately reconstruct the input data. During the model's training phase, a second encoder further processes the reconstructed output into a refined encoded form. This additional encoding step is instrumental in refining the model's detection capabilities. At the same time, the discriminator is meticulously trained to analyze the reconstructed data and assess whether it can be differentiated from the original input. The second encoder and discriminator are crucial for anomaly detection as they fine-tune the model's parameters to better recognize and reconstruct standard data patterns. It is worth emphasizing that the progression from input to the first encoder and then through the decoder mirrors the traditional autoencoder architecture. However, the integration of

an additional encoder and the discriminator further enhances the model’s performance by adding a layer of sophistication to the anomaly detection process.

The innovation of GAN-AN lies in its objective function, which strategically integrates three different loss measures:

- **Contextual Loss:** It measures the difference between the original data point (x) and the reconstructed data point ($\hat{x}$) to ensure the decoder is effectively recreating the input.
- **Encoder Loss:** Compares the initial lower-dimensional representation (z) with the one generated by the second encoder ($\hat{z}$) to ensure that important features are preserved during reconstruction.
- **Adversarial Loss:** It measures the difference between real and generated data points, to ensure that the discriminator is effective.

The effectiveness of the GAN-AN model depends on reducing these loss functions, thereby training the model to detect anomalies when a reconstructed data point markedly deviates from the standard patterns learned during training. This comprehensive objective function is crucial to the model’s success, as shown in the methodical flow of the GAN-AN process in Fig. 2.1.

Isolation Forest

Isolation means ”separating an instance from the rest.” [42] Due to their rarity and distinctiveness, anomalies are more susceptible to isolation. The isolation forest method constructs a collection of isolation trees (iTrees) for a dataset, where anomalies are characterized by shorter average path lengths within these trees. Through recursive partitioning in data-driven random trees, instances are differentiated until each stands alone. This ensures anomalies have shorter paths because they require fewer partitions and are marked by unique attribute values. Collectively, a forest of random trees yields shorter paths for particular instances, signifying possible anomalies. For a clearer understanding of how anomalous data undergoes recursive partitioning, see Fig. 2.2. In this context, the partition count can be equated to paths. Here we see that isolating the data point X_i

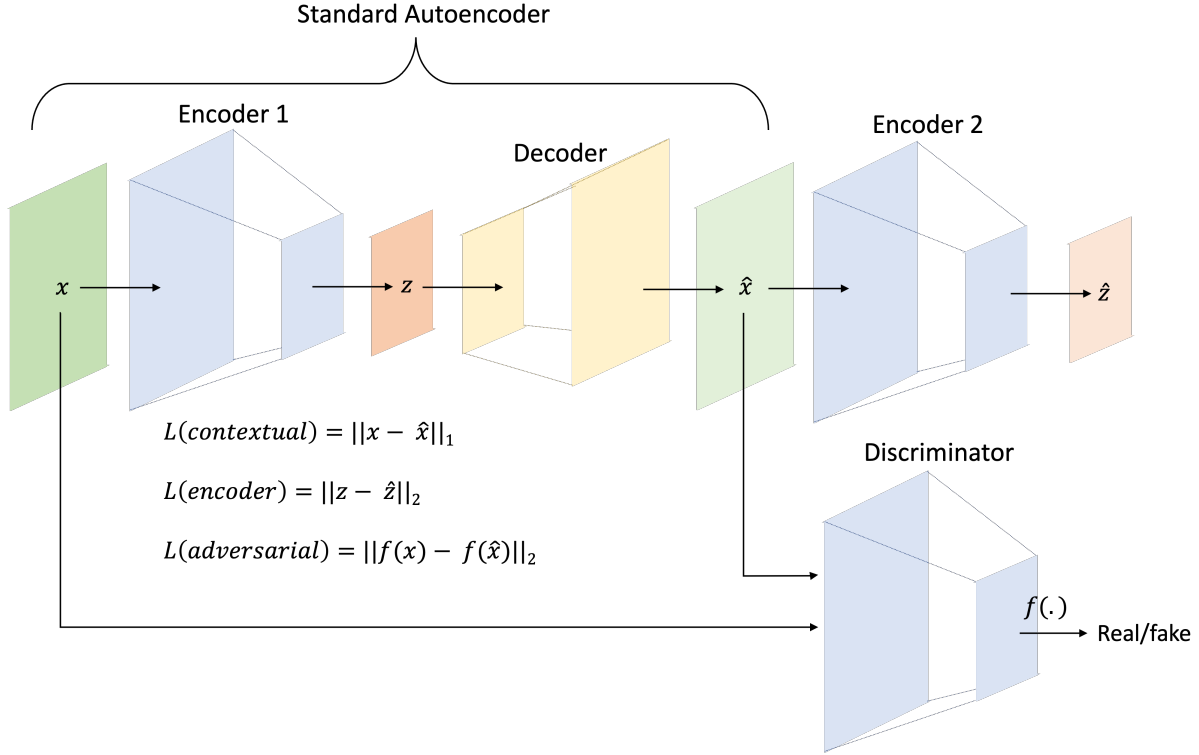


Figure 2.1: Overview of the GAN-AN approach, which contains two encoders, a decoder, and discriminator networks, employed within three sub-networks. An autoencoder which is also the generator of the model learns to recreate the original input using an encoder and a decoder. The discriminator is trained to distinguish the real data point from the fake. The second encoder comprises the reconstruction into a latent code $\hat{z}$. The objective function considers three types of losses including contextual, encoder, and adversarial loss.

requires only two partitions, thus X_i might be an anomalous data point. Further insights into the training phase are available in Algorithms 1 and 2 [42].

We employed the Python scikit-learn library [55] to train our isolation forest models. We utilized default parameters, with the contamination rate configured to "auto".

Algorithm 1 $iForest(X, t, \psi)$

Inputs: X - input data, t - number of trees, ψ - subsampling size

Output: a set of t iTrees

- 1: Initialize $Forest$
 - 2: set height limit $l = ceiling(\log_2 \psi)$
 - 3: for $i = 1$ to t do
 - 4: $X' \leftarrow sample(X, \psi)$
 - 5: $Forest \leftarrow Forest \cup iTree(X', 0, l)$
 - 6: end for
 - 7: return $Forest$
-

Algorithm 2 $iTree(X, e, l)$

Inputs: X - input data, e - current tree height, l - height limit

Output: an iTree

- 1: if $e \geq l$ or $|X| \leq 1$ then
 - 2: return $exNode\{Size \leftarrow |X|\}$
 - 3: else
 - 4: let Q be a list of attributes in X
 - 5: randomly select an attribute $q \in Q$
 - 6: randomly select a split point p from max and min values of attribute q in X
 - 7: $X_l \leftarrow filter(X, q < p)$
 - 8: $X_r \leftarrow filter(X, q \geq p)$
 - 9: return $inNode\{Left \leftarrow iTree(X_l, e + 1, l),$
 - 10: $Right \leftarrow iTree(X_r, e + 1, l),$
 - 11: $SplitAtt \leftarrow q,$
 - 12: $SplitValue \leftarrow p\}$
 - 13: end if
-

One-class Support Vector Machine

The OCSVM is an extension of the conventional Support Vector Machine (SVM), specifically designed for anomaly or outlier detection. Unlike standard SVMs, which address binary or multiclass classification tasks, OCSVM specializes in distinguishing the ‘normal’ category from possible outliers. Its objective is to derive a function f that is positive for inliers and

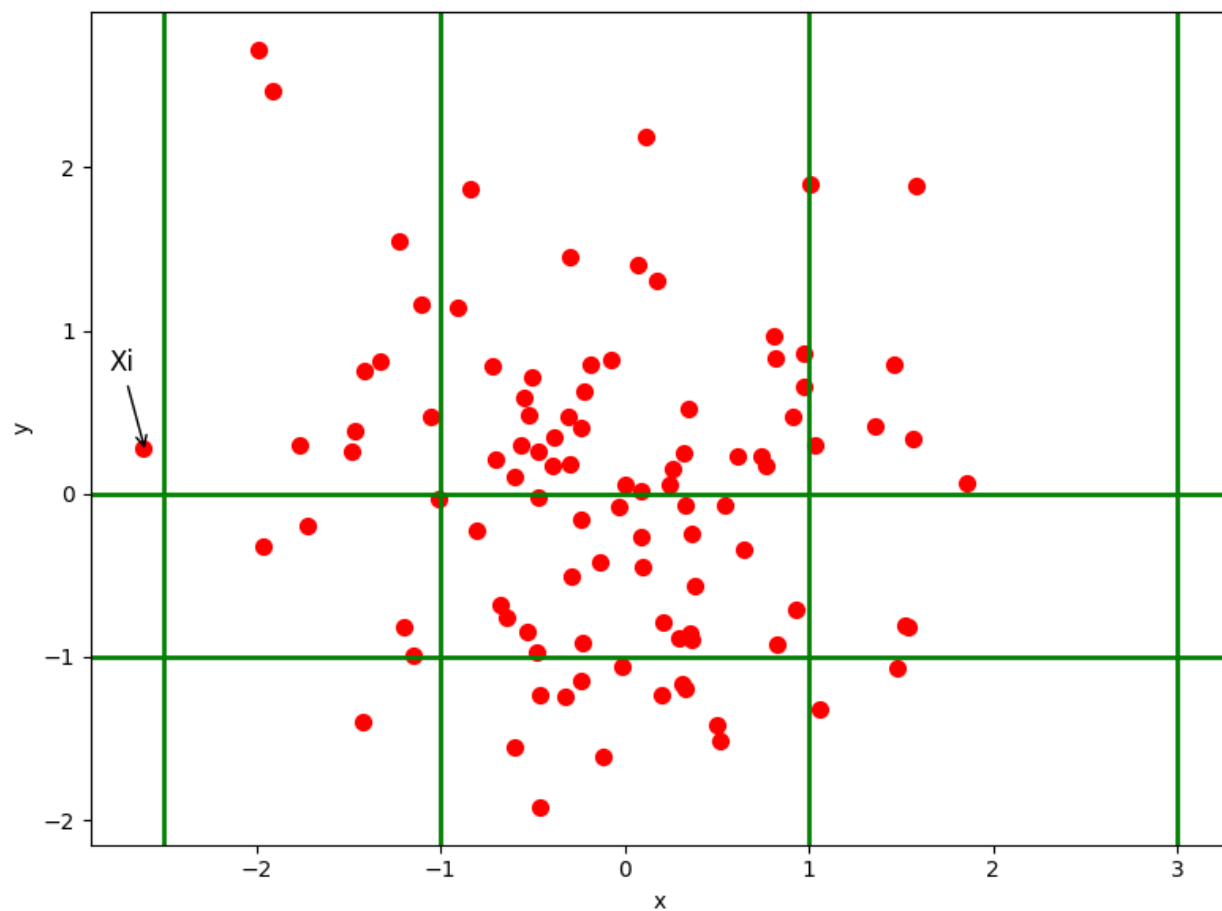


Figure 2.2: Isolating an anomalous data point using recursive splitting. The data point X_i is a probable anomaly as only two splits are needed to isolate it.

negative for outliers. To distinguish outliers from regular data, the OCSVM resolves the subsequent quadratic program [62]:

$$\begin{aligned} \min_{w \in \mathcal{F}, \xi \in \mathbb{R}^t, \rho \in \mathbb{R}} \quad & \frac{1}{2} \|w\|^2 + \frac{1}{\nu l} \sum_i \xi_i - \rho \\ \text{subject to} \quad & (w \cdot \Phi(x_i)) \geq \rho - \xi_i, \quad \xi_i \geq 0 \end{aligned}$$

where w and ρ denote the weight vector and bias of separating hyperplane, respectively and ξ_i is a slack variable.

We again employed the Python scikit-learn library [55] to train our OCSVM models. We utilized default parameters, with the kernel set to "rbf".

2.3.3 Model Evaluation

We used several commonly used metrics to evaluate and compare our models, including precision, recall, F1 score, and AUC-ROC (area under the receiver operating characteristic curve). Additionally, due the nature of class imbalance in our datasets, we specifically use PR-AUC (precision-recall area under the curve) and g-mean metrics.

Precision represents the proportion of accurately predicted positive instances out of all predicted positives and recall represents the proportion of accurately predicted positives out of all actual positives. F1 metric considers both precision and recall. That is:

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

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

where TP is the number of true positives, FP is the number of false positives and FN is the number of false negatives, and,

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}.$$

AUC is the area under the ROC curve. ROC is generated as a tradeoff between true positive rate and false positive rate [35]. On the other hand, PR-AUC can be thought of as the average of precision scores calculated for each recall threshold.

G-mean is an important metric especially for anomaly detection. It considers both recall and specificity where specificity is defined as the model’s ability to determine the negative cases correctly. That is:

$$Specificity = \frac{TN}{TN + FP}$$

where TN is the number of true negatives, and

$$G - mean = \sqrt{Recall * Specificity}.$$

2.4 Numerical Results

In this section, we first present the within-project testing results followed by cross-project testing.

2.4.1 Within-project testing

First, we evaluate and compare four anomaly detection models: autoencoder, GAN-AN, isolation forest, and OCSVM. For each model, we calculate the average performance across five datasets using five distinct evaluation metrics. To ensure the robustness of our results, we employed cross-validation. For cross-validation, we created different training and testing data groups by using 10 unique random seeds. That is, we trained and tested each model 10 separate times, with each instance using a different random split of the data. The performance of each model was then assessed using the five metrics, and the average scores and their respective standard deviations from these 10 trials, averaged across the five datasets, are presented in Table 2.3.

The comparative analysis of model performance across various metrics yields some insightful observations. While no single model consistently outperforms across all the

Table 2.3: Means and Standard Deviations of Key Metrics Across Datasets.

Model	Precision		Recall		F1 Score		G-mean		AUC-PR	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Autoencoder	72.28%	1.45%	74.83%	0.04%	73.36%	0.74%	66.33%	1.32%	80.73%	0.73%
GAN-AN	90.91%	2.03%	86.78%	2.20%	88.71%	1.80%	59.29%	12.23%	94.58%	1.22%
Isolation Forest	90.38%	1.53%	90.12%	1.03%	90.22%	0.67%	58.26%	7.83%	94.44%	0.77%
OCSVM	89.11%	1.59%	92.83%	0.00%	90.91%	0.83%	47.84%	13.20%	94.01%	0.80%
Average	85.67%	1.65%	86.14%	0.82%	85.80%	1.01%	57.93%	8.65%	90.94%	0.88%

metrics, autoencoders show noteworthy efficacy in terms of the g-mean. Nevertheless, their performance on other metrics falls short when compared to alternative models. Consequently, we compare the performance of the three models: GAN-AN, isolation forest, and OCSVM, across various metrics to focus on where the differences are more pronounced. Among these three models, GAN-AN, on average, excels in precision, g-mean, and PR-AUC. Its F1 score is not far behind those of the other two models, showing competitive robustness. On a dataset-by-dataset basis, GAN-AN frequently surpasses the other two models in all evaluated metrics. However, it is important to highlight the significantly higher variance in the g-mean observed with GAN-AN and OCSVM.

We delved deeper into the performance of each model across various datasets using boxplots to illustrate the metric scores for each type of model. Fig. 2.3 provides a visual representation of these distributions for the autoencoder, GAN-AN, isolation forest, and OCSVM models. These plots clearly highlight that the g-mean scores are notably lower in comparison to the substantially higher values observed for the other four metrics. The results also demonstrate fluctuations in performance metrics, including some outliers. Additionally, it's clear from the data that GAN-AN generally holds up well across all metrics. Notably, for one dataset, its g-mean score is close to 80%.

Further, we compare our findings with the recent state-of-the-art as presented in [24, 3]. Our models exhibit improved F1 scores and PR-AUC values. Specifically, GAN-AN, on average, achieves 20% higher F1 score and 7% higher PR-AUC compared to [24]. However, a point of concern is our models' lower g-means. It's important to note that previous studies have not reported g-means, which means we lack a direct comparison for this metric.

2.4.2 Cross-project testing

In exploring the broader effectiveness of our models, we employed cross-project testing. Isolation forest showed consistent performance across multiple datasets and evaluation metrics, making it the chosen method for this cross-project validation. This involved training the models on each individual dataset and then testing them against the other four. Similar to within-project testing, we used only non-defective data for training purposes.

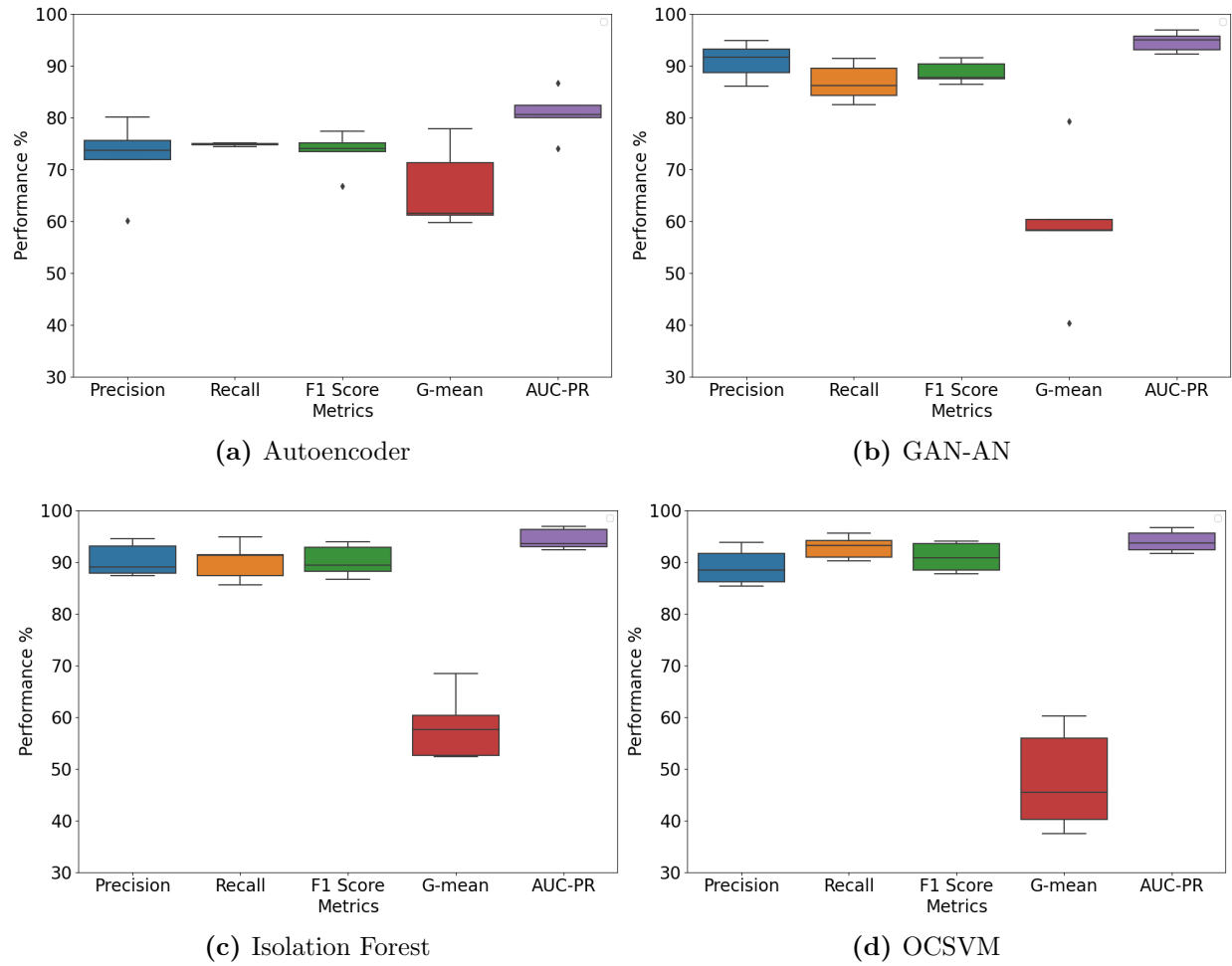


Figure 2.3: Comparison of the results over four model approaches: Autoencoder, GAN-AN, isolation forest, and OCSVM. An important observation is that the four metrics: Precision, Recall, F1 score, and AUC-PR are substantially higher than g-means. There are some outliers too especially while fitting the autoencoder model. GAN-AN performs better than isolation forest and OCSVM especially while looking at g-means

To thoroughly assess our approach, we examined the PR-AUC and the g-mean scores obtained from these cross-project tests, which are presented in Fig. 2.4. The results are organized such that each column corresponds to the performance metrics, PR-AUC and g-mean, achieved when a model, trained on one specific dataset, is then evaluated on the remaining datasets. For example, the first column of the matrix on the left shows the PR-AUC for the model trained on the KC1 dataset and subsequently tested on the others, which explains the absence of data for KC1 in that column. The matrix on the right shows the corresponding g-means.

The presented data reveals that the PR-AUC scores are impressively consistent, maintaining a range between 92-97%, suggesting a high degree of stability and resilience in model performance when subjected to cross-project evaluation. Conversely, the g-mean scores depict significant fluctuations within an approximate range of 38% to 78%.

2.5 Conclusion and Future Works

In our study, we employed GAN-AN, a new approach for detecting anomalies in the field of software defect prediction. We rigorously compared GAN-AN with three other well-known methods, autoencoder, isolation forest, and OCSVM. Our observations revealed that GAN-AN generally performs better in terms of precision, g-mean, and PR-AUC on average. Additionally, GAN-AN stands out in other metrics for specific datasets, displaying its potential for particular use cases. On the other hand, the autoencoder excelled in terms of g-mean, though our overall g-means were comparatively conservative. In essence, our collective models surpassed previous outcomes on average, indicating progress in the field of anomaly detection for software defect prediction. This suggests that selecting the most suitable model could depend on dataset characteristics, and a trade-off might be necessary based on the specific metric of interest.

As a part of future work, we recommend enhancing the robustness of our models through meticulous parameter tuning. By refining the GAN-AN method, more sophisticated data features can be identified that can improve predictive accuracy. This may help deepen the understanding of the GAN-AN approach in the context of software defect prediction.

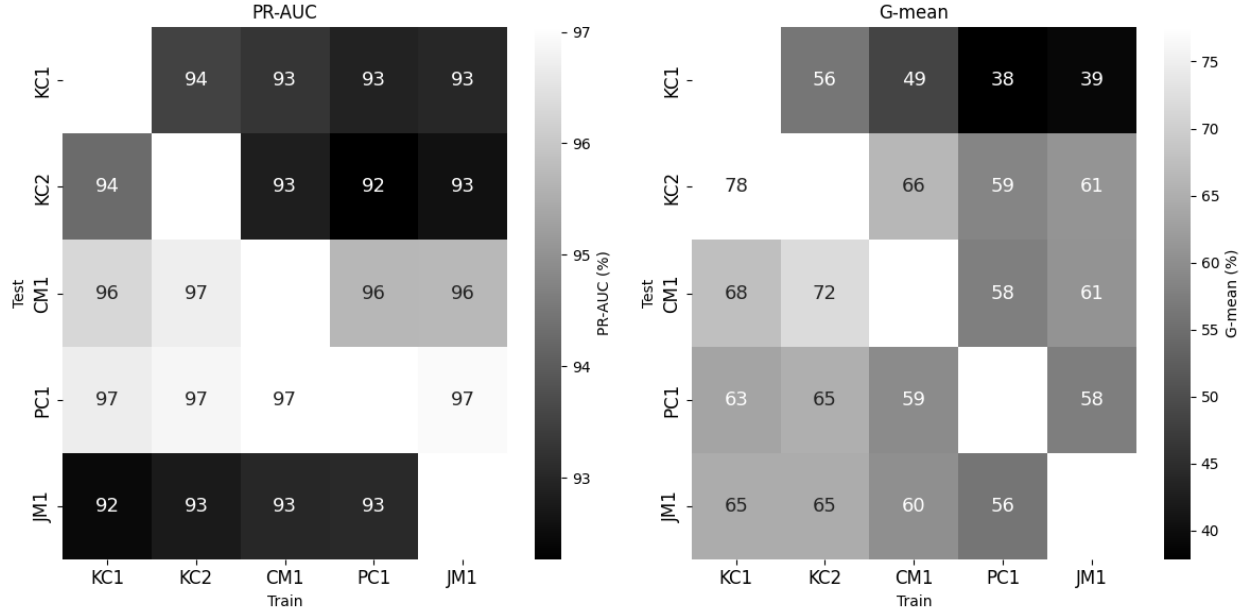


Figure 2.4: Cross-project evaluation metrics: For each dataset, models underwent training and were subsequently assessed on the four other datasets. The PR-AUC and g-mean scores are summarized in the tables provided. Specifically, the matrix on the left presents the PR-AUC scores. For example, the first column presents the PR-AUC when models trained on the KC1 dataset (horizontal axis) were evaluated against the others (vertical axis), with the first row blank to indicate the exclusion of the KC1 dataset in this phase. Similarly, the matrix on the right showcases the g-mean results under the same conditions.

Chapter 3

Does Graph Neural Network Improve Software Defect Prediction?

3.1 Introduction

Software defects can have various origins and can occur at different stages throughout the software development process. One of the primary causes of these defects can be attributed to developer oversights [46]. This can occur due to a lack of attention to detail or inadequate testing during the development phase. Other contributing factors include incomplete or unclear requirements, coding errors, and insufficient quality control procedures. It is essential to identify and address these issues early on in the development process to minimize the impact of defects on the final product. Extensive research has been conducted in the literature to address these issues, and various techniques have been employed.

In the evolving field of software defect prediction, conventional methods like random forests and logistic regression coexist with more advanced deep learning techniques, such as Temporal Convolutional Networks and Convolutional Neural Networks. These advanced methods are developed to handle the increasing complexity of software systems, aiming to offer a more nuanced understanding and identification of defects. As software becomes more complex, the advancement in defect prediction methods highlights the need for varied and flexible techniques in identifying software issues.

Furthermore, in response to the increasing intricacies of software, Graph Neural Networks (GNNs) are being explored as an innovative approach [22, 79]. They excel in handling graph-structured data and are particularly well-suited for uncovering the complex interactions within software development, such as the links between commits, developers, and repositories. GNNs offer a new perspective in understanding and analyzing the intricate interdependencies that may be inherent in data structures.

Our study aims to investigate the potential of Graph Neural Networks (GNNs) in improving the detection of software defects. We took an innovative approach by transforming our data into a tri-partite graph structure [20]. This structure comprises of three types of nodes, each representing an entity namely developers, commits, and repositories, and each node has associated feature vectors. We then utilized these graphs to train Relational Graph Convolutional Networks (R-GCNs) [61]. R-GCNs are a type of GNN known for their ability to handle heterogeneous types of graphs, such as the tri-partite graphs used in our study. These networks can process and interpret complex and varied relationships to capture inherent interdependencies within these entities, making them a promising tool for predicting software defects.

We evaluate our approach using 11 large-scale JiTReliability open-source projects [70]. We utilize the existing defective commit labels in these datasets to train binary supervised classification models that are specifically designed to detect and identify defective commits. The main focus of our research is to create tri-partite graphs for each of the 11 datasets and apply R-GCNs to them. By utilizing these advanced techniques, we aim to assess the effectiveness of using graph structures in analyzing software defect data across various datasets. This innovative approach demonstrates our commitment to explore and leverage the latest advancements in computational technology. The main contributions of this study are as follows:

- We introduce a novel tri-graph structure that captures the interdependencies among the three key entities involved in software development: developers, commits, and repositories. These interdependencies are not necessarily captured in traditional models, making our proposed graph a meaningful way to formalize these interdependencies.

- We leverage the proposed graph structure to model GNN models for early detection of defective commits, potentially empowering software developers to promptly address and remediate issues.
- Finally, we compare the results of our proposed approach against the baseline logistic regression model.

The remainder of the manuscript is organized as follows. In Section 3.2, we conduct a comprehensive review of the relevant literature concerning Neural networks used for JIT-SDP and further highlight the current research gaps and our contributions. In Section 3.3, we provide an overview of the datasets used, present the problem formulation, describe the models employed, and introduce the evaluation metrics utilized. The testing results for each of the 11 datasets are presented in Section 3.4. Finally, Section 3.5 concludes the paper by summarizing the findings and outlining potential avenues for future research.

3.2 Background and Related Work

Extensive research has been conducted in the domain of software defect prediction. Various approaches and methodologies have been employed to improve the accuracy and efficiency of these processes. A notable trend in recent years is the increasing use of deep learning frameworks, which provide significant advancements in just-in-time (JIT) defect prediction.

One of the key contributions in this field includes the Deep Learning-based defect localization approach introduced by [57]. This technique utilizes a sequence-to-sequence model to analyze the naturalness of code, assigning a suspiciousness score to each line in a code change. The source code lines at the top of the list are considered to be a possible defect location. This tool is designed to provide developers with early-stage detection of buggy lines in their code. The results of this study surpassed the previous state of the art by over 11% in accuracy. Next, [9, 10] introduced and later enhanced the use of Temporal Convolutional Networks (TCNs) for JIT defect prediction. These studies leverage a comprehensive set of features based on source code metrics, demonstrating the effectiveness of TCNs in predicting software defects at the commit level. The enhancement included

hierarchical attention layers, which achieved an accuracy of over 95% and an F1 score of over 91%. This approach underscores the importance of combining product and software metrics extracted from commits of software projects along with their evolution. Complementing these studies, [34] introduced the DeepJIT framework, which extracts features from both commit messages and code changes. This framework, tested on various software projects, showed an improvement of about 9% to 13% highlighting the potential of deep learning in analyzing complex data sets for defect prediction. Further, [64] expanded this field by introducing a graph convolution neural network model specifically designed for abstract syntax trees (AST). The proposed model uses the information of the nodes and edges from the AST of the source code. The architecture of the proposed model was claimed to be adaptive to the unique structures of different software projects. Next, [81] introduced an approach that focuses on analyzing changes in the ASTs of source code before and after a change. This method extracts AST change sequences and then maps them into numeric vectors by using word embedding technology. Their method, which incorporates AST Change Embedding with a gated hierarchical model, integrates traditional manual features with semantic features derived from source code changes, showcasing an effective method for allocating testing resources. Their results showed an improvement of 4% for within-project testing and 2.4% for cross-project testing over the baseline results. On the front of vulnerability detection, [68] explores the applicability of graph neural networks in learning the nuances of source code. That is, whether signatures of vulnerabilities in source code can be learned from their graph representations and studying the relationships between nodes and edges. The idea behind this study is to present a large dataset already tagged as vulnerable or non-vulnerable to a learning-based model so that the model can learn from the properties that differentiate vulnerable code from healthy code.

Parallel to these developments in deep learning, traditional model-fitting techniques have continued to play a crucial role in software change classification and prediction. Notably, random forests and regression models have been prominent, as highlighted by the survey in [77]. Further, [36, 37] employed random forest to both within-project and cross-project testing to achieve the median area under the curve (AUC) values of approximately 78% and 71%. Support vector machines, used by [38] for classifying software changes, further,

underline the diversity in approaches. Next, [48] aimed to predict the failure probability of a change to the source code using logistic models. Their approach was based on various predictors, including size in lines of code, diffusion of the change, developer experience, and the type of change. Moreover, a systematic survey in [77] indicated that ensemble learning techniques in JIT-SDP could attain F1 scores ranging from 25% to 75%. Consequently, unsupervised models have gained popularity in order to address some of the limitations posed by traditional classification methods. These limitations mainly include the skewness of the data distributions and the lack of availability of labeled data. For instance, [44] explored the use of anomaly detection for fine-grained JIT defect prediction, achieving an AUC of 71%. Another study in [76] proposed to transform the binary classification problem into an anomaly detection problem to effectively tackle the class imbalance problem and partially mitigate the issue of limited labeled data. This study also achieved an AUC of approximately 71%. Furthermore, resampling methods for balancing data, involving undersampling overrepresented classes or oversampling underrepresented ones, have been explored in multiple studies ([45, 69, 72]). While unsupervised models offer solutions to the challenges of imbalanced data, it is crucial to recognize that their predictive performance may not consistently match the levels achieved by their supervised counterparts.

In summary, this landscape of software defect prediction presents a rich variety of methodologies, each contributing uniquely to the advancement of the field [77]. These diverse studies collectively contribute to a deeper understanding of software defect prediction. They showcase a spectrum of approaches, from innovative deep learning frameworks to tried-and-tested traditional model fitting techniques, each playing a vital role in advancing the field.

3.2.1 Our contribution

Our study proposes a new approach for learning from a tripartite-graph structure that captures the interdependencies among developers, commits, and repositories. There are three types of nodes in this graph, each having a set of feature vectors, and we use a GNN algorithm, specifically an R-GCN model, to learn from its structure. By encapsulating these interdependencies, our approach offers a unique perspective in the field.

Our approach represents a significant departure from previous methods. While past research has primarily focused on analyzing graph representations to understand the relationships between nodes and edges within the source code itself ([68]), our methodology diverges notably. We concentrate on metadata that is *external* to the code, as opposed to the *internal* structure and nuances of the source code explored in earlier studies. This distinction places our work in a unique position, emphasizing the importance of external metadata in understanding and predicting software defects.

3.3 Methods

3.3.1 Datasets

We conducted experiments on 11 open-source projects across multiple programming languages and fields [70]. By using these popular datasets, we can establish a basis for comparing our results with the results of prior studies and assess the validity of our findings. The summarized statistics for each dataset, including the project age in years (Age), kilo lines of code (KLOC), number of changes (#Changes), and defect ratio (Bug %) are presented in Table 3.1. It is important to note that all 11 datasets exhibit some level of class imbalance. Among them, HBase stands out as the most imbalanced dataset, with only 5% of commits being defective. On the other hand, Hadoop appears to be the most balanced dataset, with over 26% of commits being classified as defective. The diverse class imbalance across the datasets allows for the further examination of the generalizability of the results. Each of these 11 datasets consists of 14 features categorized into five distinct dimensions, listed in Table 3.2. We have opted to use these features for consistency based on their successful track record in traditional defect prediction research.

3.3.2 Problem Formulation

Consider dataset D , which contains n software commits. Let commit c be represented by feature vector $X_c = [x_{c1}, x_{c2}, \dots, x_{cp}]$, $c = 1, 2, \dots, n$, where p is the number of features associated with each commit. Next, each developer class d can be represented as $A_d =$

Table 3.1: Statistics of the datasets used in our study.

Project	Description	Age(year)	KLOC	#Changes	Bug %
ActiveMQ	High-performance messaging server.	13.45	38	10,213	16.15%
Ant	Java-based build tool.	8.75	339	14,387	6.08%
Camel	Open source integration framework.	11.53	75	38,563	9.04%
Derby	A pure Java, standards-based relational database engine.	9.42	1350	8,268	15.74%
Hadoop	Open source distributed computing system.	11.59	102	16,084	26.67%
HBase	A distributed, scalable, big data store.	5.03	413	10,509	5.21%
IVY	A project dependencies managing tool.	3.29	135	2,880	16.18%
JCR	An implementation of the Content Repository for Java Technology API.	8.82	38	8,651	6.44%
JMeter	An application to load test applications and measure performance.	15.89	264	16,341	7.54%
LOG4J2	A logging library for Java.	5.37	85	10,690	4.79%
LUCENE	High-performance, full-featured text search engine library.	15.93	183	31,240	7.03%

Table 3.2: Traditional Features Characterized by 5 Dimensions

	Name	Definition
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directions
	NF	Number of modified files
	Entropy	Distribution of modified code across each line
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	Fix	Whether a change is a defect fix
History	NDEV	Number of developers that changed modified files
	AGE	Average time between last and current change
	NUC	Number of unique changes to the modified files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

$[a_{d1}, a_{d2}, \dots, a_{dq}]$, $d = 1, 2, \dots, m$, where q is the number of features associated with each developer class and m is the number of developer classes. Finally, each repository r can be represented as $B_r = [b_{r1}, b_{r2}, \dots, b_{rs}]$, $r = 1, 2, \dots, o$, where s is the number of features associated with each repository and o is the number of repositories. Furthermore, let $Y = [y_1, y_2, \dots, y_n]$ denote the target vector, representing the status (i.e., labels) of the commits, where y_c equals 1 if commit c is defective and equals 0, otherwise.

To extract insights from the input data, we analyze it as a tripartite graph $G = (V, E)$, where V is the set of all vertices and E is the set of all edges (Fig. 3.1). In Fig. 3.1, the tripartite graph consists of three columns of nodes, each representing a specific entity with their own feature vectors. Let's explain each column in detail:

- The first column of nodes represents developer class, where each developer class is associated with its own feature vector $A_d = [a_{dq}]$. Here, d denotes the number of developer classes and q is the number of features associated with each developer class, such as developer experience.
- The second column of nodes represents the commits made by the developers, along with their own feature vector $X_c = [x_{cp}]$. Here, c denotes the number of commits and p is the number of features associated with each commit, such as number of files added/deleted during that commit.
- Finally, the last column of nodes represents the repositories where the commits are made, also with their feature vectors $B_r = [b_{rs}]$. Here r denotes the number of repositories and s is the number of features associated with each repository, such as the size of the repository before the change.

We represent our dataset as a graph in order to capture the complex relationships between developers, commits, and repositories. From there, we can use a graph neural network model to analyze the data. This can potentially enhance the accuracy and efficacy of our classification model. For instance, the experience of a developer can impact the quality of the code written which in turn can affect the overall quality of the software being developed.

The vertices in V are divided into three disjoint subsets: V^D , V^C , and V^R representing developers, commits, and repositories, respectively. We derive the developer and repository

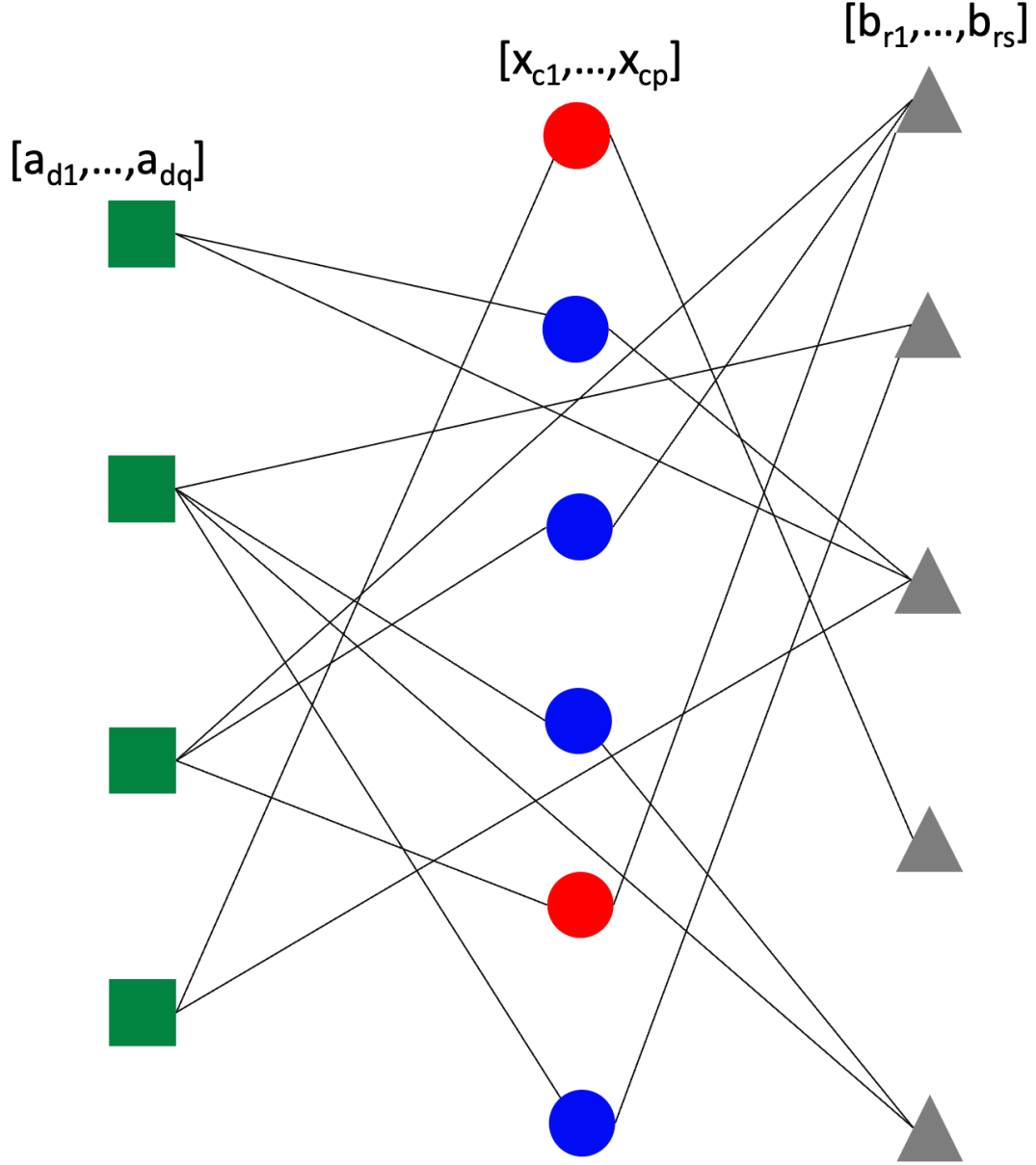


Figure 3.1: The first column (left) of the nodes denotes developers with feature vectors a_{dq} , the middle column represents commits with feature vectors x_{cp} and the last column (on the right) represents the repositories to which commit has been made with feature vectors b_{rs} .

vertices from the feature vector X_c that contains features a_{dq} and b_{rs} representing the developer class and the repository feature, respectively, for each commit. Each edge in the tripartite graph G can only connect nodes across different subsets [20]. An edge $(v_d, v_c) \in E$ connects a developer vertex, $v_d \in V^D$, to a commit vertex, $v_c \in V^C$, if the commit was made by the developer belonging to that class. An edge $(v_c, v_r) \in E$ connects a commit vertex, $v_c \in V^C$, to a repository vertex, $v_r \in V^R$, if the commit is made to a file represented by the corresponding repository feature. And, an edge $(v_d, v_r) \in E$ connects a developer vertex, $v_d \in V^D$, to a repository vertex, $v_r \in V^R$, if a developer belonging to that class commits to a file represented by the corresponding repository feature.

Next, we establish a direct correspondence between the structure of the tripartite graph illustrated in Fig. 3.1 and a sample graph we constructed using our data.

In Fig. 3.2, we present a sample graph that we created using Python’s NetworkX library using our dataset. For our data, the generation of nodes involved the utilization of three distinct features: transaction ID (for each commit), developer class (based on developer experience), and lines of code in the file before the change (i.e., file size prior to the change). This graph is formed using a two-hop neighborhood approach, wherein nodes within two degrees of separation from a randomly selected transaction node are examined. In other words, the graph is generated by selecting a transaction node at random and identifying its directly connected nodes (i.e., one hop away), as well as the nodes connected to those nodes (i.e., two hops away).

Note that this is just a sample graph and for demonstration purposes, we limited the number of neighbors to a maximum of 10. Within the graph, commits are represented by circles, with blue circles denoting non-defective commits and red circles signifying defective ones. The triangles in the graph represent the file size before the change. This feature is utilized to represent the characteristics of a repository, as it plays a crucial role in assessing the likelihood of introducing defects. For instance, larger files may exhibit higher complexity and consequently be more prone to defects, whereas smaller files may be more manageable and less likely to introduce errors. The developers are represented by squares. Each square represents a specific developer class.

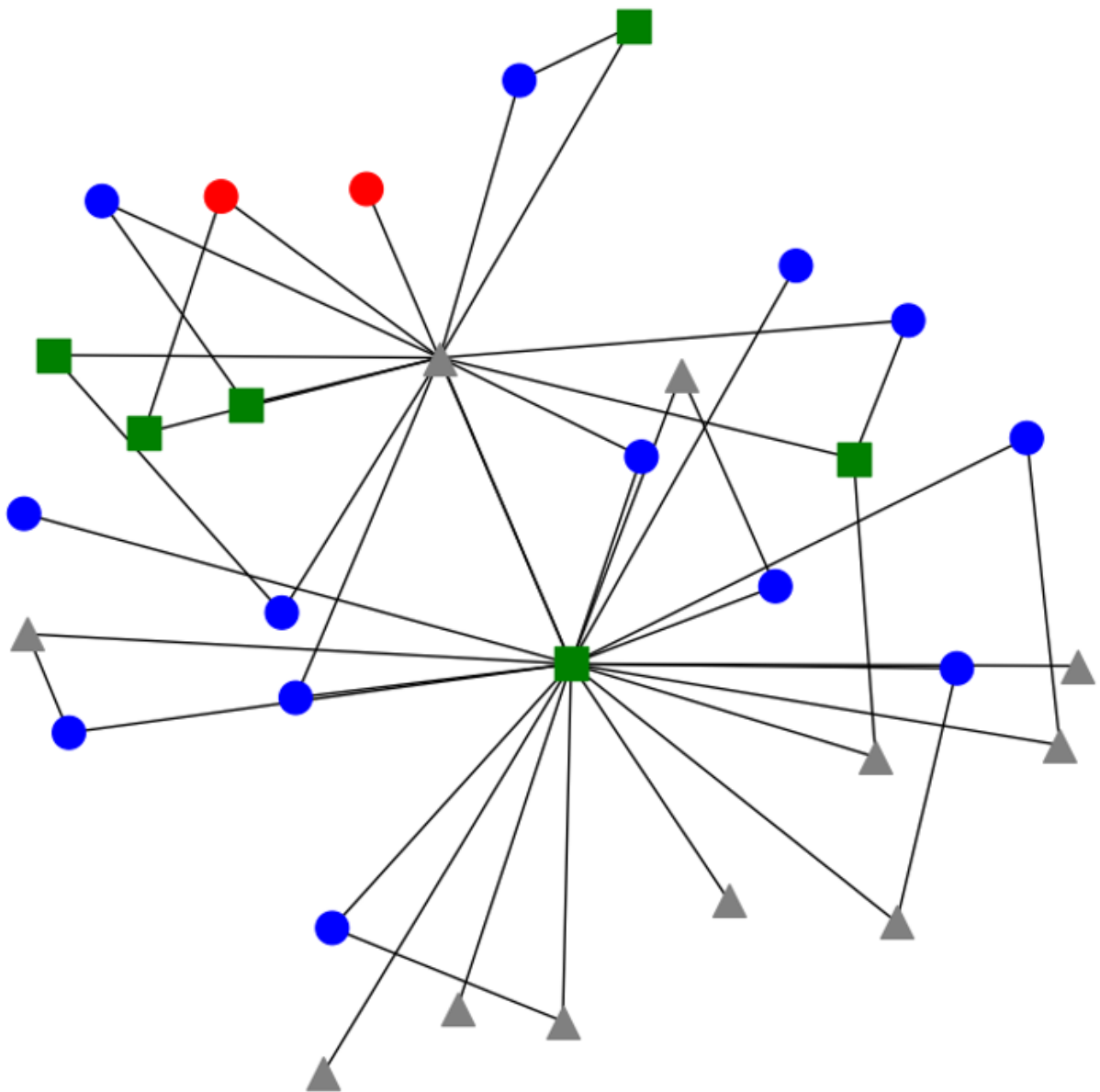


Figure 3.2: Circles = commits; blue = non-defective, red = defective, Triangle-Up = lines of code in the file before the change (representative of the size of repository), Square = developer

A direct correspondence can be established between the sample graph we constructed using our data and the tripartite graph structure depicted in Fig. 3.1. Let us delve into further detail regarding this correspondence. We have three distinct types of nodes in the graph illustrated in Fig. 3.2: commits, developers, and the file size before the change. Similarly, in the tripartite graph showcased in Fig. 3.1, three entities are present: commits, developers, and repositories. In our constructed graph, we represent a transaction as a node, which is connected to two types of nodes: developer nodes and file size nodes.

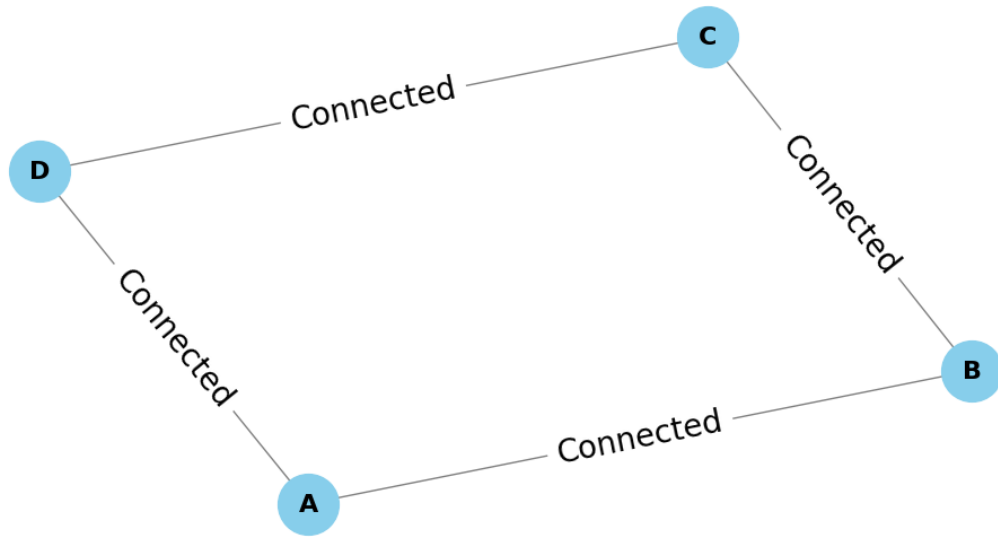
- The transaction nodes correspond to individual commits that are being made.
- In order to assess developer experience, we arranged developers into classes based on experience intervals of 10 units. For example, developers with 1-10 units of experience fall into one class, while those with 11-20 units are in another. This classification effectively places each developer within a specific experience range, simplifying the assessment of their contribution to the commits.
- In our tripartite graph, the third set of nodes denotes repositories. Although we don't categorize this entity explicitly, we include a measure like 'lines of code in the file before the change' to reflect the repository size prior to a commit. This approach, while focusing on a single repository, allows us to incorporate and analyze the repository entity's influence in our study.

We now move forward to explore and examine the models utilized in our research.

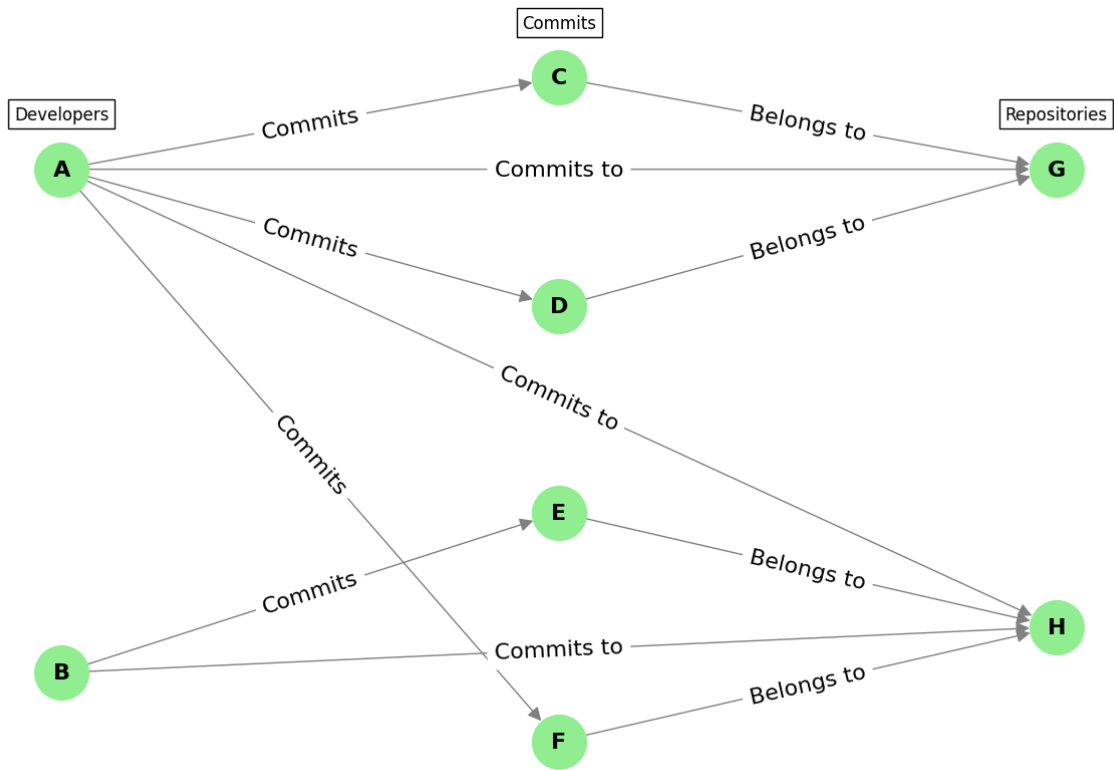
3.3.3 Models

Relational Graph Convolutional Network

R-GCNs are a type of neural network architecture designed for processing graph-structured data, particularly graphs with relational information. We use an R-GCN to represent the complex relational structure in our tripartite graph data. They are highly effective when it comes to handling heterogeneous graph data [61]. This is due to their ability to model a variety of relation types simultaneously. The significance of our R-GCN model lies in its



(a) GCN



(b) R-GCN

Figure 3.3: Comparison of a simple GCN structure to an R-GCN structure. GCN model is more suitable for data structures involving a single relation type. However, R-GCNs are better suited for multi-relational type data.

ability to discern and leverage the intricate patterns of connectivity inherent to tripartite graphs and its flexibility through hyperparameter tuning.

R-GCNs build upon the concept of Graph Convolution Networks (GCNs), offering an enhanced framework suitable for more complex relational data. While GCNs are ideal for graphs with a single type of edge, as illustrated in Fig. 3.3a, where nodes are simply *connected*, R-GCNs come into play when multiple relationships exist among nodes, much like the multi-relational setup in our tri-partite graph described in Section 3.3.2. For instance, as we see in Fig. 3.3b, developer A *commits* the commit C, which in turn *belongs to* repository G, thus establishing that A also *commits to* repository G.

An R-GCN distinguishes itself by processing various types of relationships in a graph through specialized aggregation functions. These functions consider the nature of node connections, enabling the R-GCN to learn nuanced relational representations. In a standard GCN, node representations are updated through a uniform process: gathering neighbor embeddings, merging them, and applying a linear transformation. This involves a singular weight matrix for each layer ($W^{(l)}$), shared among all nodes, which combines the embeddings followed by an activation function. The updated node features are computed as per Kipf and Welling’s method [39]:

$$h_i^{l+1} = \sigma \left(\sum_{j \in N_i} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)} \right),$$

where N_i are the neighbors of node i , c_{ij} is a normalization factor, and $W^{(l)}$ is the shared projection matrix for layer l .

On the other hand, R-GCNs demand a more customized approach due to the existence of multiple edge types. They use separate transformation matrices for every connection type within the graph layers. This means that in R-GCNs, aggregation and projection are specific to each relation, semantically aligning different types of edges after summation. The representation updating formula for R-GCNs can be expressed as:

$$h_i^{l+1} = \sigma \left(W_0^{(l)} h_i^{(l)} + \sum_{r \in R} \sum_{j \in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)} \right),$$

where r denotes each unique relationship type, and every GCN layer hosts corresponding weight matrices (W 's). The terms in the equation have the following meanings:

- $h_i^{(l+1)}$: The updated feature vector of node i at layer $l + 1$.
- σ : A non-linear activation function.
- $W_0^{(l)}$: The self-connection weight matrix for node i at layer l .
- $h_i^{(l)}$: The feature vector of node i at layer l .
- $W_r^{(l)}$: The relationship-specific weight matrix for type r at layer l .
- $c_{i,r}$: A normalization factor related to node i and relationship type r .
- R : The set of all relationship types in the graph.
- N_i^r : The neighbors of node i connected by relationship type r .

This framework enables the capture of complex interactions between different types of relationships and neighboring nodes through dual aggregation.

Our model's training involves a label smoothing loss function, enhancing generalization by mitigating overconfidence in predictions. We utilized conventional optimization techniques and employed early stopping to curb overfitting.

3.3.4 Evaluation Metrics

We evaluated the model using standard metrics: Precision, Recall, F1 score, and Precision-Recall Area Under the Curve (PR-AUC).

Precision

Precision represents the proportion of accurately predicted positive instances out of all predicted positives. That is:

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

where TP is the number of true positives and FP is the number of false positives.

Recall

Recall represents the proportion of accurately predicted positives out of all actual positives.

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

where FN is the number of false negatives.

F1 score

F1 metric considers both precision and recall. That is:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}.$$

Precision-Recall Area Under the Curve

AUC and PR-AUC are both metrics used to evaluate the performance of binary classification models. AUC measures the overall ability of the model to distinguish between positive and negative classes, while PR-AUC specifically focuses on the model’s ability to identify true positives while minimizing false positives. AUC is less sensitive to class imbalance than PR-AUC, making PR-AUC a better choice for problems where the positive class is rare.

3.4 Results and Discussion

The performance of the R-GCN model on our 11 datasets is summarized in Table 3.3. The results using a standard logistic regression model (baseline) is presented in Table 3.4. Contrary to expectations, the R-GCN model did not outperform the baseline model. This outcome prompts a critical analysis of the underlying factors that might contribute to this result.

Firstly, the relationships among features in our datasets may not be as complex as hypothesized. Logistic regression’s comparative performance suggests that the feature interactions may be linear or that a linear model is sufficient to capture the essential patterns for the prediction task. If the signal in the data is inherently straightforward,

Table 3.3: GNN performance metrics

Dataset	Precision	Recall	F1-score	PR-AUC
Active	28.6%	67.9%	34.6%	71.7%
Ant	11.7%	70.9%	17.8%	72.6%
Camel	16.7%	74.2%	23.8%	76.6%
Derby	29.9%	72.0%	36.2%	76.4%
Hadoop	8.7%	64.9%	13.2%	65.8%
HBase	42.0%	66.5%	44.4%	72.6%
IVY	30.0%	70.0%	36.3%	74.4%
JCR	14.3%	80.5%	20.9%	82.7%
JMeter	11.6%	62.4%	17.4%	63.6%
LOG4J2	10.1%	74.0%	16.1%	75.4%
LUCENE	13.8%	78.3%	20.5%	80.4%

Table 3.4: Logistic regression performance metrics

Dataset	Precision	Recall	F1-score	PR-AUC
Active	30.82%	57.98%	40.23%	71.16%
Ant	11.94%	55.09%	19.62%	70.30%
Camel	19.50%	63.75%	29.86%	75.31%
Derby	30.58%	62.42%	40.98%	74.75%
Hadoop	9.29%	51.18%	15.69%	66.44%
HBase	47.81%	61.08%	53.59%	74.56%
IVY	28.91%	61.14%	39.02%	71.46%
JCR	19.19%	69.93%	30.05%	82.23%
JMeter	15.34%	57.54%	24.21%	69.56%
LOG4J2	11.33%	63.12%	19.18%	76.82%
LUCENE	17.68%	63.35%	27.61%	76.97%

the additional complexity of an R-GCN model may not translate to better performance. This means that the graph structure that we implement may not be efficient in capturing the interdependencies among the three entities: developers, commits, and repositories. Furthermore, we have individual datasets for each repository and used a pseudo feature to represent the repository entity. This may not accurately capture the feature of the entity as we had anticipated.

Secondly, the tri-partite graph we constructed seems too sparse since the number of edges in the graph is either similar to or lower than the number of nodes in our graph. The sparsity of the graphs could significantly impact the R-GCN’s performance. Graph neural networks, including R-GCN, typically rely on rich neighborhood information for each node to learn effective representations. If the graph is sparse, the model may struggle to aggregate meaningful information from the neighbors, potentially resulting in subpar performance outcomes. Sparse graphs limit the message-passing capabilities, which is a cornerstone of graph neural network functionality.

As part of the study, an experiment was conducted to reduce the dimensionality of non-defective commits. The aim was to balance the number of classes and improve the model’s ability to generalize. However, this strategy did not significantly enhance the performance metrics of the R-GCN model. This suggests that class imbalance might not be the main factor affecting the model’s effectiveness. Furthermore, to ascertain whether the integration of heterogeneity in the model construction had a substantive effect, an alternative approach was adopted by constructing a homogeneous graph. Consequently, standard Graph Convolutional Networks (GCN) [39] and Graph Attention Networks (GAT) [71] were employed. However, these adaptations also did not yield fruitful results.

The findings suggest that for our dataset, the complexity of R-GCN models does not enhance prediction accuracy. This emphasizes the need to match data characteristics with the complexity of the chosen model and to validate the efficacy of advanced techniques against simpler ones. The unexpected performance of R-GCNs, which did not surpass that of basic logistic regression, prompts a reevaluation of the presumption that complexity equates to effectiveness in software defect prediction. Our research demonstrates that simple methods

can sometimes outperform more complex learning techniques, prompting a reevaluation of these strategies.

3.5 Conclusion and Future Work

Our research delved into the intricate world of software defect prediction with a novel approach. We aimed to investigate whether a graph-based model, in particular the R-GCN, could offer a better understanding and more precise forecast of software defects by capturing the complex relationships among developers, commits, and repositories. We created a complex tri-partite graph for each dataset to model relationships, but our advanced R-GCN model didn't perform better than a simpler logistic regression model in defect prediction. This study has shed new light on the ongoing debate surrounding software defect prediction. Our findings suggest that in some cases, simpler methods can be just as effective, if not more so, than complex ones. This study not only makes a significant contribution to the field of software defect prediction, but also prompts us to reconsider our approach to leveraging technology in problem-solving across various domains. It highlights the importance of understanding the specific nature of the data and the problem at hand before choosing or designing models, and reinforces the idea that the complexity of a model does not always equate to its effectiveness.

As a part of future work, we propose experimenting with various graph structures, potentially blending graph-based models like R-GCNs with traditional methods to leverage their combined strengths. Addressing the sparsity of our tri-partite graph is another critical area, possibly by incorporating additional data sources or inferring missing relationships.

Conclusion

In this thesis, we have explored several innovative approaches to enhance Just-In-Time Software Defect Prediction (JIT-SDP), each contributing to a deeper understanding and more effective prediction of software defects.

During our research, we focused on the integration of graph-based features, specifically Graph Defect Ratio (GDR), which is derived from a tripartite graph connecting developers, commits, and repositories. Our study highlights the GDR as a potent alternative to a broad spectrum of traditional features in software defect prediction, underscoring the need for further exploration to fully leverage its advantages in enhancing modeling efficiency in software engineering. The adoption of GDR could streamline the modeling process, offering a more simplified yet equally effective approach by encapsulating the intricate interdependencies within software development into a single, comprehensive metric.

In addition, we explored anomaly detection techniques for software defect prediction using the GAN-AN model. This model is a generative adversarial network that has been specifically designed for anomaly detection. Our findings indicate that GAN-AN performs better than traditional methods like autoencoders, isolation forests, and OCSVM, particularly with regards to precision and PR-AUC. This suggests that selecting the right model based on dataset characteristics and desired metrics is crucial. Additionally, our work highlights the potential for more advanced methods and parameter tuning in the future.

Our research on Relational Graph Convolutional Networks (R-GCNs) has shown that these complex models do not necessarily perform better than simpler logistic regression models in predicting defects. This finding challenges the belief that complexity is always linked to effectiveness and highlights the importance of considering the specific nature of the data and the problem context.

As a direction for future work, we suggest delving deeper into the use of graph-based techniques. This includes experimenting with various graph structures and combining graph-based models with traditional methods.

Moreover, tackling the issue of graph sparsity and integrating more intricate data characteristics present valuable opportunities. Additionally, exploring cutting-edge graph neural networks, such as the Inductive Heterogeneous Graph Neural Network [18], holds great potential for advancing our research and uncovering new insights. Such endeavors will enhance our comprehension of graph-based methods in software defect prediction, opening up new avenues in this domain.

Bibliography

- [1] (2014). Shellshock: All you need to know about the bash bug vulnerability. [2](#), [23](#)
- [2] (2020). The heartbleed bug. [2](#), [23](#)
- [3] Afric, P., Sikic, L., Kurdija, A. S., and Silic, M. (2020). Repld: Source code defect prediction as anomaly detection. *Journal of Systems and Software*, 168:110641. [27](#), [40](#)
- [4] Ahmed, M., Mahmood, A. N., and Hu, J. (2016a). A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31. [24](#)
- [5] Ahmed, M., Mahmood, A. N., and Islam, M. R. (2016b). A survey of anomaly detection techniques in financial domain. *Future Generation Computer Systems*, 55:278–288. [24](#)
- [6] Akcay, S., Atapour-Abarghouei, A., and Breckon, T. P. (2019). Ganomaly: Semi-supervised anomaly detection via adversarial training. In *Computer Vision–ACCV 2018: 14th Asian Conference on Computer Vision, Perth, Australia, December 2–6, 2018, Revised Selected Papers, Part III 14*, pages 622–637. Springer. [24](#), [32](#)
- [7] Akimova, E. N., Bersenev, A. Y., Deikov, A. A., Kobylkin, K. S., Konygin, A. V., Mezentsev, I. P., and Misilov, V. E. (2021). A survey on software defect prediction using deep learning. *Mathematics*, 9(11):1180. [26](#)
- [8] Albahli, S. (2020). A deep ensemble learning method for effort-aware just-in-time defect prediction. *Applied Computer Science*, 16(3):5–15. [28](#)
- [9] Ardimento, P., Aversano, L., Bernardi, M. L., Cimitile, M., et al. (2020). Temporal convolutional networks for just-in-time software defect prediction. In *ICSOFIT*, pages 384–393. [46](#)
- [10] Ardimento, P., Aversano, L., Bernardi, M. L., Cimitile, M., and Iammarino, M. (2022). Just-in-time software defect prediction using deep temporal convolutional networks. *Neural Computing and Applications*, 34(5):3981–4001. [46](#)

- [11] Aversano, L., Cerulo, L., and Del Grosso, C. (2007). Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 19–26. [3](#), [7](#)
- [12] Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761. [3](#), [7](#)
- [13] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. (2011). Don’t touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. [3](#), [7](#)
- [14] Boehm, B. and Basili, V. R. (2001). Defect reduction top 10 list. *Computer*, 34(1):135–137. [5](#)
- [15] Briand, L. C., Basili, V., and Hetmanski, C. J. (1993). Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering*, 19(11):1028–1044. [3](#), [7](#)
- [16] Bryan, J. and Moriano, P. (2023). Graph-based machine learning improves just-in-time defect prediction. *Plos one*, 18(4):e0284077. [5](#), [6](#), [7](#)
- [17] Cabral, G. G. and Minku, L. L. (2022). Towards reliable online just-in-time software defect prediction. *IEEE Transactions on Software Engineering*, 49(3):1342–1358. [6](#)
- [18] Cai, D., Qian, S., Fang, Q., Hu, J., and Xu, C. (2023). User cold-start recommendation via inductive heterogeneous graph neural network. *ACM Transactions on Information Systems*, 41(3):1–27. [66](#)
- [19] Chalapathy, R. and Chawla, S. (2019). Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407*. [24](#)

- [20] Chartrand, G., Lesniak, L., and Zhang, P. (2010). *Graphs & digraphs*, volume 39. CRC press. [8](#), [45](#), [54](#)
- [21] Chen, L., Fang, B., and Shang, Z. (2016). Software fault prediction based on one-class svm. In *2016 International Conference on Machine Learning and Cybernetics (ICMLC)*, volume 2, pages 1003–1008. IEEE. [27](#)
- [22] Chen, Z., Komrmusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. (2019). Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959. [45](#)
- [23] D’Ambros, M., Bacchelli, A., and Lanza, M. (2010). On the impact of design flaws on software defects. In *2010 10th International Conference on Quality Software*, pages 23–31. IEEE. [1](#)
- [24] Ding, Z., Mo, Y., and Pan, Z. (2019). A novel software defect prediction method based on isolation forest. In *2019 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE)*, pages 882–887. IEEE. [27](#), [40](#)
- [25] Dynkin, B. and Dynkin, B. (2017). Derivative liability in the wake of a cyber attack. *Alb. LJ Sci. & Tech.*, 28:23. [2](#), [23](#)
- [26] D’Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17:531–577. [27](#)
- [27] Gaudreault, J.-G., Branco, P., and Gama, J. (2021). An analysis of performance metrics for imbalanced classification. In *International Conference on Discovery Science*, pages 67–77. Springer. [14](#)
- [28] Gong, L., Jiang, S., and Jiang, L. (2019). Tackling class imbalance problem in software defect prediction through cluster-based over-sampling with filtering. *IEEE Access*, 7:145725–145737. [28](#)
- [29] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press. [24](#)

- [30] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, 27. [24](#), [32](#)
- [31] Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc. [29](#)
- [32] Hastie, T., Tibshirani, R., Friedman, J. H., and Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer. [14](#)
- [33] Ho, T. K. (1995). Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE. [13](#)
- [34] Hoang, T., Dam, H. K., Kamei, Y., Lo, D., and Ubayashi, N. (2019). Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE. [4](#), [47](#)
- [35] Hossin, M. and Sulaiman, M. (2015). A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5(2):1. [13](#), [38](#)
- [36] Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., and Hassan, A. E. (2016). Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106. [2](#), [4](#), [5](#), [13](#), [23](#), [47](#)
- [37] Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. (2012). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773. [2](#), [4](#), [5](#), [6](#), [7](#), [23](#), [47](#)
- [38] Kim, S., Whitehead, E. J., and Zhang, Y. (2008). Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering*, 34(2):181–196. [2](#), [4](#), [47](#)
- [39] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*. [58](#), [63](#)

- [40] Krasner, H. (2021). The cost of poor software quality in the us: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*. 2
- [41] Li, N., Shepperd, M., and Guo, Y. (2020). A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology*, 122:106287. 28
- [42] Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2008). Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. 24, 33, 35
- [43] Liu, J., Zhou, Y., Yang, Y., Lu, H., and Xu, B. (2017). Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 11–19. IEEE. 5
- [44] Lomio, F., Pascarella, L., Palomba, F., and Lenarduzzi, V. (2022). Regularity or anomaly? on the use of anomaly detection for fine-grained just-in-time defect prediction. In *30th IEEE/ACM International Conference on Program Comprehension (ICPC 2022)*, volume 1, page 05. 5, 27, 48
- [45] Malhotra, R. and Kamal, S. (2019). An empirical study to investigate oversampling methods for improving software defect prediction using imbalanced data. *Neurocomputing*, 343:120–140. 5, 48
- [46] Mays, R. G. (1990). Applications of defect prevention in software development. *IEEE Journal on Selected Areas in Communications*, 8(2):164–168. 44
- [47] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320. 29
- [48] Mockus, A. and Weiss, D. M. (2000). Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180. 2, 4, 5, 23, 48
- [49] Moshtari, S., Santos, J. C., Mirakhorli, M., and Okutan, A. (2020). Looking for software defects? first find the nonconformists. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 75–86. IEEE. 26

- [50] Nassif, A. B., Talib, M. A., Nasir, Q., and Dakalbab, F. M. (2021). Machine learning for anomaly detection: A systematic review. *IEEE Access*, 9:78658–78700. [24](#)
- [51] Neela, K. N., Ali, S. A., Ami, A. S., and Gias, A. U. (2017). Modeling software defects as anomalies: A case study on promise repository. *J. Softw.*, 12(10):759–772. [27](#)
- [52] Ogheneovo, E. E. et al. (2014). On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 2(14):1. [1](#)
- [53] Pandey, S. K. and Tripathi, A. K. (2021). An empirical study toward dealing with noise and class imbalance issues in software defect prediction. *Soft Computing*, 25(21):13465–13492. [28](#)
- [54] Paul, R., Turzo, A. K., and Bosu, A. (2021). Why security defects go unnoticed during code reviews? a case-control study of the chromium os project. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1373–1385. [2](#)
- [55] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830. [14](#), [35](#), [37](#)
- [56] Pelayo, L. and Dick, S. (2007). Applying novel resampling strategies to software defect prediction. In *NAFIPS 2007 - 2007 Annual Meeting of the North American Fuzzy Information Processing Society*, pages 69–72. [5](#)
- [57] Qiu, F., Gao, Z., Xia, X., Lo, D., Grundy, J., and Wang, X. (2021). Deep just-in-time defect localization. *IEEE Transactions on Software Engineering*, 48(12):5068–5086. [46](#)
- [58] Qiu, Y., Liu, Y., Liu, A., Zhu, J., and Xu, J. (2019). Automatic feature exploration and an application in defect prediction. *IEEE Access*, 7:112097–112112. [13](#)
- [59] Raymond, E. S. and O’Reilly, T. (1999). *The Cathedral and the Bazaar*. O’Reilly & Associates, Inc., USA, 1st edition. [1](#)

- [60] Sayyad Shirabad, J. and Menzies, T. (2005). The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada. [27](#), [29](#)
- [61] Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., and Welling, M. (2018). Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*, pages 593–607. Springer. [45](#), [56](#)
- [62] Schölkopf, B., Williamson, R. C., Smola, A., Shawe-Taylor, J., and Platt, J. (1999). Support vector method for novelty detection. *Advances in neural information processing systems*, 12. [24](#), [37](#)
- [63] Shepperd, M., Song, Q., Sun, Z., and Mair, C. (2013). Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering*, 39(9):1208–1215. [27](#)
- [64] Šikić, L., Kurdija, A. S., Vladimir, K., and Šilić, M. (2022). Graph neural network for source code defect prediction. *IEEE Access*, 10:10402–10415. [47](#)
- [65] Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5. [6](#)
- [66] Sofaer, H. R., Hoeting, J. A., and Jarnevich, C. S. (2019). The area under the precision-recall curve as a performance metric for rare binary events. *Methods in Ecology and Evolution*, 10(4):565–577. [14](#)
- [67] Song, Q., Guo, Y., and Shepperd, M. (2018). A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering*, 45(12):1253–1269. [24](#)
- [68] Suneja, S., Zheng, Y., Zhuang, Y., Laredo, J., and Morari, A. (2020). Learning to map source code to software vulnerability using code-as-a-graph. *arXiv preprint arXiv:2006.08614*. [47](#), [49](#)

- [69] Tan, M., Tan, L., Dara, S., and Mayeux, C. (2015). Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE. [5](#), [48](#)
- [70] Tian, Y., Li, N., Tian, J., and Zheng, W. (2020). How well just-in-time defect prediction techniques enhance software reliability? In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 212–221. IEEE. [45](#), [49](#)
- [71] Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y., et al. (2017). Graph attention networks. *stat*, 1050(20):10–48550. [63](#)
- [72] Wang, S. and Yao, X. (2013). Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443. [5](#), [24](#), [48](#)
- [73] Wu, R., Zhang, H., Kim, S., and Cheung, S.-C. (2011). Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. [27](#)
- [74] Yang, X., Lo, D., Xia, X., and Sun, J. (2017). Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206–220. [4](#), [13](#)
- [75] Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., and Leung, H. (2016). 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*, pages 157–168. [5](#), [28](#)
- [76] Zhang, S., Jiang, S., and Yan, Y. (2022). A software defect prediction approach based on bigan anomaly detection. *Scientific Programming*, 2022. [5](#), [27](#), [48](#)
- [77] Zhao, Y., Damevski, K., and Chen, H. (2023). A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, 55(10):1–35. [4](#), [5](#), [13](#), [47](#), [48](#)
- [78] Zheng, W., Shen, T., Chen, X., and Deng, P. (2022). Interpretability application of the just-in-time software defect prediction model. *Journal of Systems and Software*, 188:111245. [7](#)

- [79] Zhou, C., He, P., Zeng, C., and Ma, J. (2022). Software defect prediction with semantic and structural information of codes based on graph neural networks. *Information and Software Technology*, 152:107057. [4](#), [45](#)
- [80] Zhu, K., Zhang, N., Ying, S., and Zhu, D. (2020). Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. *IET Software*, 14(3):185–195. [2](#), [4](#)
- [81] Zhuang, W., Wang, H., and Zhang, X. (2022). Just-in-time defect prediction based on ast change embedding. *Knowledge-Based Systems*, 248:108852. [47](#)

Vita

Aradhana Soni completed her Bachelor of Arts in Economics at the University of Delhi in 2009 and subsequently embarked on a career in the industry as an actuarial analyst/manager. In 2018, she returned to academia, earning a Master of Science in Mathematics and Statistics from East Tennessee State University in 2020. She then pursued her PhD in Industrial and Systems Engineering at the University of Tennessee, graduating in 2024. Her doctoral research centered on the application of machine learning techniques for the detection of defective commits in software repositories. Her scholarly interests are broad and multifaceted, encompassing areas such as machine learning, graph-based learning, software defect prediction, and the exploration of advanced computational models in data analysis.