OPEN ACCESS

Int. Jr. of Contemp. Res. in Multi. *PEER-REVIEWED JOURNAL* *Volume 4 Issue 3 [May- Jun] Year 2025*

Research Article

Leveraging Network Representation Methods to Optimize Software Defect Prediction

Sweta Mehta[1*], Pankaj K. Goswami[1], K. Sridhar Patnaik[2]
[1] Department of CSE, Sarala Birla University, Ranchi, Jharkhand, India
[2] Department of CSE, Birla Institute of Technology, Mesra, Ranchi, Jharkhand, India

Corresponding Author: * Sweta Mehta

DOI: https://doi.org/10.5281/zenodo.15768408

Abstract

The importance of Software Defect Prediction (SDP) is well-recognized due to its ability to identify potential defects early in the software development lifecycle. Traditional SDP approaches that rely on metrics such as code complexity and coupling often fall short in capturing the intricate interrelationships and interactions typical of large-scale software systems. To design a more efficient and accurate SDP model, it is essential to better represent the structural dependencies within software. Network-based graphical representations—such as call graphs and class dependency networks—offer a promising means to capture these complex relationships and hidden patterns. Call graphs illustrate function-level interactions by mapping caller-callee relationships, while class dependency networks represent dependencies between software modules. This study investigates the effectiveness of call graphs and class dependency networks in developing a highly accurate SDP framework. The evaluation involves ten Java-based open-source projects and ten machine learning classifiers. Results show that call graphs outperform class dependency networks in most cases, with improvements in AUC ranging from 2.9% to 8.94%. This performance gain is attributed to the call graphs' superior ability to model intricate relationships between software components—an essential factor in defect prediction. Among the classifiers tested, the Generative Adversarial Network (GAN) emerged as the most effective, achieving an AUC of 0.91 and an accuracy of 92.5%.

Manuscript Information

- **ISSN No:** 2583-7397
- **Received:** 29-05-2025
- **Accepted:** 25-06-2025
- **Published:** 29-06-2025
- **IJCRM:**4(3); 2025: 539-548

- **Plagiarism Checked:** Yes
- **Peer Review Process:** Yes

How to Cite this Article

Mehta S, Goswami PK, Patnaik KS. Leveraging Network Representation Methods to Optimize Software Defect Prediction. Int J Contemp Res Multidiscip. 2025;4(3):539-548.

Access this Article Online

www.multiarticlesjournal.com

KEYWORDS: Network Representations, Software Defect Prediction, PROMISE Repository, Class Dependency Networks, Call Graphs

INTRODUCTION

Software testing is known to be one of the most time-consuming and expensive phases in the software development life cycle (SDLC). It plays a key role in ensuring the quality of the final product, which is why improving testing efficiency is important to make better use of time and resources [22]. One way to achieve this is through SDP. SDP helps identify parts of the software that are likely to have bugs early in the development process. By doing this, it allows testing teams to focus their efforts on specific parts of the software, making testing more efficient and reducing the workload and costs for everyone involved. Predicting which modules are likely to be faulty also helps developers during future updates or maintenance by making them more cautious with those parts of the code. This targeted testing approach not only improves software quality but also reduces the overall cost of development [5, 14]. Since these risky modules are identified early, fewer defects reach the testing stage, and there's also a lower chance of introducing new bugs during maintenance.

Traditional SDP methods often use software metrics—like code complexity and coupling—to build prediction models. These metrics work mostly at the class level and describe how complicated or interconnected parts of the code are. While helpful, they fall short when dealing with today's increasingly complex software systems, as they can't fully capture the deeper relationships between software components [21, 3]. As a result, prediction models based only on these metrics often perform poorly, which has led researchers to explore newer, more advanced methods. These include better ways of representing the code and using powerful machine learning techniques to more accurately predict defect-prone areas, while keeping costs low. Building strong SDP models can make software more reliable overall [10].

Some recent research has used network-based techniques—especially those inspired by Social Network Analysis—to better understand how different parts of the software are connected. This approach can improve defect prediction performance. Most of these studies have used class dependency networks, which provide a broad view of how different classes in the software depend on each other. However, this focus has left other useful graph-based representations, like call graphs, relatively unexplored. To address this gap, our study investigates how using call graphs—another type of network that maps how functions call one another—can improve the accuracy and cost-effectiveness of defect prediction models.

2. Existing Research

In recent years, SDP has become a popular topic in software engineering. Many researchers have developed different SDP models by combining various techniques. These techniques often include software metrics like static code metrics and code churn, along with network-based representations such as module networks. Static code metrics are commonly used and focus only on the structure of the source code [3, 29].

For example, Singh et al. [11] used the NASA AR1 dataset, which includes many software features, to test different regression and machine learning models. Their decision tree-based model performed better than others, offering both high accuracy and easy interpretation. Zimmermann et al. [24] found that "change bursts" (sudden large changes in code) were more effective at finding defects than traditional metrics like code complexity and churn. Gupta et al. [34] analyzed 54 projects using a large dataset and showed that models performed better when trained on datasets with no overlaps and evaluated with effort-aware measures that consider the amount of work needed to find defects.

Social Network Analysis (SNA) is another technique gaining attention in software engineering. For example, Wolf et al. [26] studied the developer communication networks using SNA to predict build failures. While their work focused on build issues, our research shifts the focus to identifying defect-prone modules. Studies like Ma et al. [16] found that SNA-based models performed well, especially for predicting defects within the same project or across versions. Similarly, Nguyen et al. [27] showed that combining SNA with code metrics improved model performance. These results highlight the benefits of using a mix of software metrics, data science, and machine learning in SDP.

Boucher et al. [23] analyzed different software metrics to find patterns between defect-prone modules and their features. They also looked at ways to set useful thresholds for these metrics, finding that methods like the Alves ranking and ROC curves were effective. Ulan et al. [24] proposed an unsupervised learning approach using weighted metrics, where probability theory was used to calculate scores. However, their study focused only on a small set of object-oriented metrics, which limits its application to more diverse software projects.

Many past studies have relied heavily on traditional software metrics. While helpful to some extent, these metrics often ignore the deeper structural relationships between software modules [31]. Recent research has started using network-based representations to better capture these complex dependencies. A major study by Qu et al. [35] used a Class Dependency Network and k-core decomposition to rank classes by their likelihood of being defective. Zhou et al. [17] combined class dependency networks with abstract syntax trees and network embedding to extract structural and semantic information. Antal et al. [13] proposed a model using function-level hybrid metrics, but their work was limited to JavaScript projects, highlighting the need to validate such models across other programming languages.

Raamesh et al. [1] proposed a hybrid LSTM model enhanced with optimization algorithms to detect and fix defects, using datasets from Firefox and Bugzilla. They evaluated their model using mean squared error. Ponnala & Reddy [12] used an ensemble method combining Random Forest, SVM, and LightGBM, focusing on function-level details like method complexity and caller-callee relationships. Their model achieved an ROC of 0.853 but was tested only on one project (Broadleaf Commerce), indicating a need for wider testing. Kumar & Venkatesan [27] showed that GANs could help improve SDP, but their research included very few projects, again calling for broader validation. Alqarni & Aljamaan [10] used an ensemble model that combined AdaBoost with GAN to balance an imbalanced dataset and improve prediction. Their results showed the advantages of GAN-based models, but since they relied solely on software metrics, they didn't fully address the complex interactions between software modules.

In summary, while existing research has made good progress in software defect SDP to fully understand the structural and dependency-based causes of defects. This highlights the need for improved SDP models that can better capture the complex interactions between software components.

3. METHODOLOGY

3.1 Software Metrics

Software metrics are numerical values that represent different characteristics of a software system. In this study, we focus on two main types of metrics commonly used in SDP: traditional software metrics and network metrics based on SNA.

3.1.1 Traditional Software Metrics

Traditional software metrics are used to measure features like complexity, size, and dependencies in a software system. These metrics help developers understand the overall structure and potential problem areas in the code. Some of the earliest and most well-known metrics were introduced by Halstead and McCabe [10, 32], which focused on understanding the basic structure of the code. As software development shifted toward object-oriented programming, a new set of metrics called CK metrics [25] was introduced to better reflect object-oriented features such as inheritance, cohesion, and coupling. These traditional metrics can be grouped into a few categories:

Size Metrics: Estimate the size of the software, usually by counting lines of code.

Complexity Metrics: Measure how complex the code is—for example, cyclomatic complexity counts the number of independent paths through the code.

Code Churn Metrics: Track how often a piece of code changes over time, which can indicate instability or areas needing attention.

Object-Oriented Metrics: Focus on object-oriented features like how classes are connected (coupling), how tightly related the parts of a class are (cohesion), and how inheritance is used. These give insight into design quality and help predict which parts are more likely to have defects.

3.1.2 Network Metrics

Network metrics use graph-based models to represent the software and better understand how different parts of it interact. In these models, software components (like functions or classes) are shown as nodes, and their interactions (like function calls or dependencies) are shown as edges connecting them. This helps visualize and analyze complex relationships within the software. These network metrics are inspired by SNA [21] and are applied here using call graphs and class dependency networks (as shown in Table 1 of the study). They help identify critical modules in the software based on how they are connected. Some important network metrics used in this study include:

Degree Centrality: Measures how many direct connections a node has. A module with high degree centrality interacts with many others, indicating it's crucial to the software's functioning.

Betweenness Centrality: Identifies nodes in the network that act as connections between different parts of the network. These modules help control the flow of information and are important for detecting potential defects.

Clustering Coefficient: Measures how tightly connected a group of nodes is. A high clustering value may suggest increased complexity, which can make defects more likely.

By using these network metrics, we gain a better understanding of the software's internal structure, which helps in accurately identifying defect-prone modules.

3.2 Network Representations

In this study, we represent software as a network where the nodes are software components (like functions or classes), and the edges represent the connections or interactions between them. These network-based views help in understanding how different parts of the software are linked, making it easier to maintain, optimize, and improve the system. We focus on two types of network representations: Call Graphs and Class Dependency Networks.

3.2.1 Call Graph

A Call Graph is a network that shows how functions or methods in a program call each other [6, 18]. In this graph: Nodes represent functions or methods while the edges show the direction of function calls—from the calling function (caller) to the called function (callee).

Call graphs can be Static i.e created by analyzing the source code without running it. This gives a complete view of all possible calls in the code, or dynamic i.e built while the code runs, showing only what happens during specific executions. This study uses static call graphs because they provide a detailed overview of the function dependencies across the codebase. The idea is that the more function calls a class has, the more complex it becomes, which increases the risk of defects during development or maintenance. By analyzing these dependencies, we can better predict which parts of the software are more likely to have defects [20].

3.2.2 Class Dependency Network

A Class Dependency Network represents the relationships between classes in an object-oriented program [4]. In this network: Nodes are classes or interfaces while the edges represent structural connections, such as inheritance, usage, or import relationships between the classes.

This type of graph gives a high-level view of how the software is organized and how different classes depend on each other. It's useful for understanding how changes in one part of the code might affect others. Since defects in one module can often affect connected modules, this type of representation is commonly used to help predict where future bugs might appear.

3.3 ML Algorithms Utilized

The goal of this study is to build models that can predict whether a class in the software is defect-prone or not—a binary classification problem. To do this, we use and compare several machine learning algorithms from different categories [6]. These include simple models, ensemble methods, and deep learning approaches, to ensure a diverse and well-rounded evaluation. Here's a breakdown of the types of classifiers used:

Tree-based classifiers (e.g., Decision Trees): Make decisions by following paths in a tree structure.

Probabilistic classifiers (e.g., Naive Bayes): Predict based on probability.

Instance-based classifiers (e.g., K-Nearest Neighbors): Compare new data with similar past examples.

Linear classifiers (e.g., Logistic Regression): Use linear boundaries to separate classes.

Deep Learning models (e.g., Multi-Layer Perceptron, LSTM): Learn complex, non-linear relationships in the data.

Ensemble methods (e.g., Random Forest, Bagging, Gradient Boosting): Combine multiple models to improve accuracy.

Each model has its own strengths: Naive Bayes and Logistic Regression are simple and easy to interpret. Random Forest and Bagging + Decision Trees work well with unbalanced data and prevent overfitting. K-Nearest Neighbors finds patterns without needing strong assumptions. Gradient Boosting focuses on fixing previous mistakes to improve predictions. Multi-Layer Perceptron (MLP) captures complex patterns using two hidden layers. Long Short-Term Memory (LSTM) is especially good at learning long-term dependencies in data [33]. Because defect datasets often have fewer defective classes (i.e., they're imbalanced), we also include models like LightGBM (a fast-boosting method) and GANs, which are neural networks that can generate synthetic data to improve model training. These help in handling imbalance and improving prediction accuracy.

3.4 Dataset Description

The proposed framework is tested using Java-based software projects. The projects and their defect data are sourced from the PROMISE repository [2], which provides defect information at the class level. Table 3 summarizes the key details of these projects, including the total number of classes, the number of defective classes, and the percentage of defective classes. For each project, the defect data includes twenty software metrics for every class, along with a column indicating whether the class has defects. A specific version of ten different projects is chosen, with defect percentages ranging from 8.97% to 63.57%. This wide range of defect percentages ensures diversity in the dataset, which helps in developing a more generalized and robust defect prediction model.

Table 3 : Summary of the software projects

Project	Software Version	Total Classes	Defective Classes (%)	Project	Software Version	Total Classes	Defective Classes (%)
Synapse	1.2	256	87 (33.98%)	Ant	1.7	745	166 (22.28%)
Tomcat	6.0	858	77 (8.97%)	Camel	1.6	965	189 (19.58%)
Velocity	1.6	229	78 (34.06%)	Ivy	2	352	40 (11.36%)
Poi	3	442	281 (63.57%)	jEdit	4.1	312	80 (25.64%)
Xalan	2.6	885	412 (46.44%)	Lucene	2.4	340	203 (59.7%)

Based on the dataset in Table 3, the study evaluates defect prediction models using ten different Java-based software projects, each with varying levels of defects. These projects, ranging from Ant to Xalan, represent a wide range of defect percentages, from a low of 8.97% (Tomcat 6.0) to a high of 63.57% (Poi 3). The total number of classes across these projects varies, with Ant containing 745 classes and Poi having the largest number at 442 classes. The percentage of defective classes also varies significantly; for instance, Lucene 2.4 has a high defect rate of 59.7%, while Ivy 2 has a relatively low defect percentage of 11.36%. The diversity in defect percentages and project sizes provides a comprehensive dataset for training and evaluating defect prediction models, ensuring that the resulting framework can generalize well across different types of software systems.

3.5 Overview of Experimental Setup

This study models the relationships and interactions between software components using network representations of the source code. Two types of network representations—class dependency networks and call graphs—are used to capture class-level dependencies at varying levels of detail and granularity. Section 3.2 provides a detailed explanation of these representations. To gain insights from the structural data, network metrics are employed in this study, enabling a

comparison of their performance in defect prediction against traditional metrics. Additionally, a cost analysis is conducted to evaluate whether the use of network metrics improves the cost-effectiveness of the model.

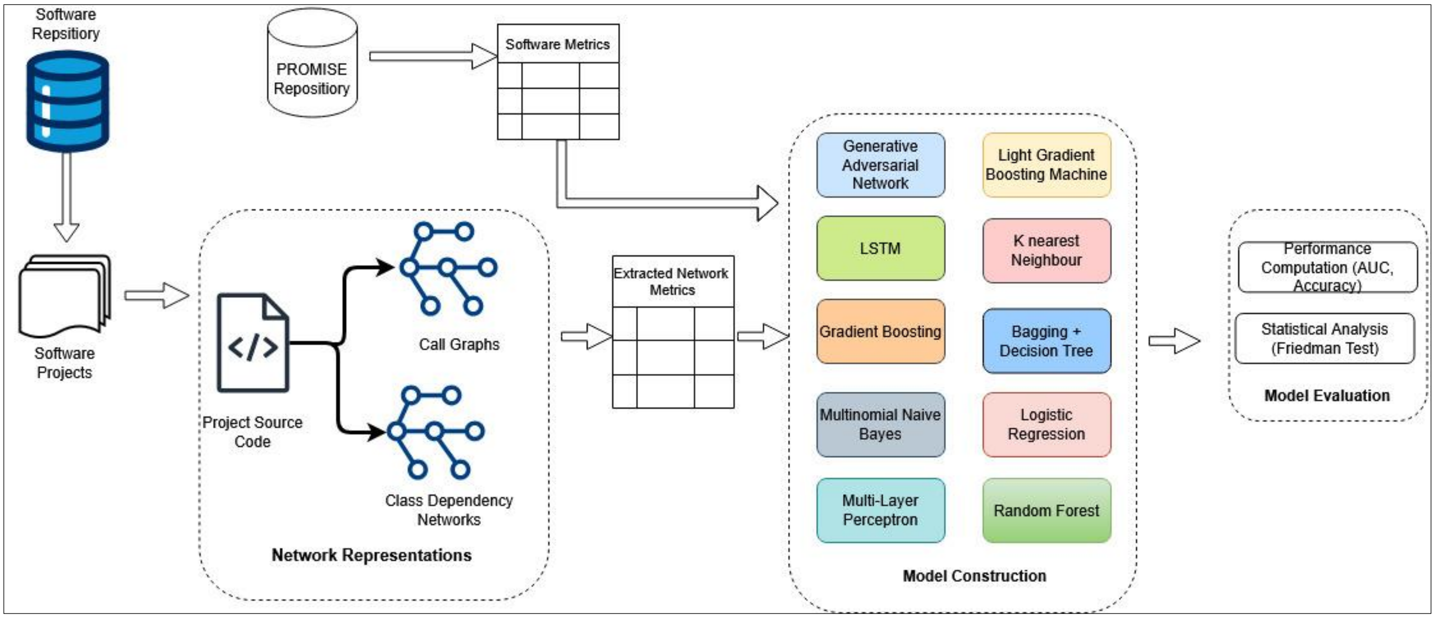

Figure 1: Overview of the experimental framework

The experimental process begins by acquiring the source code of the selected projects from their respective codebases, followed by retrieving defect data from the well-known PROMISE repository [11] in the software engineering field. Static representations of the Call Graph and Class Dependency Network are generated by parsing the source code of each project using the Understand tool, resulting in two graphical representations for every project. The next step involves extracting network metrics from the generated call graphs and class dependency networks using the Ucinet tool. Once these metrics are obtained, the defect dataset for training the SDP model is created by linking the class-level defect labels from the PROMISE repository with the corresponding class-level network metrics. By correlating the network metrics with the presence or absence of defects, the resulting dataset captures the structural relationships at different granularities for both call graphs and class dependency networks.

The dataset is used to first train the developed SDP model, followed by testing, where the network metrics serve as the input features and the defect information acts as the target or output label. To compare the impact of network metrics derived from call graphs and class dependency networks, separate SDP models are built using both types of networks. The performance of these models is evaluated based on key metrics like Area Under the Curve (AUC) and accuracy, and the results are further compared to models using traditional software metrics. Seven different machine learning classifiers are applied to identify the best-performing model in the context of network-based metrics. To validate the findings, a statistical test, such as the Friedman test, is used to assess whether the differences in model performance are statistically significant. An overview of the major stages involved in developing this framework is presented in Figure 1.

4. Experimental Results

The results presented in the tables reflect the performance of various SDP models, focusing on the use of Call Graph-based, Class Dependency Network-based, and Traditional Software Metrics-based models. The analysis covers the accuracy and Area Under the Curve (AUC) for different machine learning classifiers, such as MNB, LR, RF, KNN, GB, BAGD, MLP, LSTM, LightGBM, and GAN, across a set of software projects.

The Call Graph-based SDP models exhibit strong performance in both accuracy (Taable 4) and AUC (Table 5), with LightGBM and GAN classifiers consistently achieving the highest values. For instance, the Ant project has an accuracy of 93.46% and an AUC of 0.92 when using GAN. Similarly, LightGBM also shows promising results, with an accuracy of 90.35% and AUC of 0.91. On average, LightGBM and GAN outperform other classifiers across all projects, highlighting their potential in identifying defect-prone software components with high precision.

Table 4: Performance metrics (Accuracy) for SDP models (Call Graph-based)

Projects	Accuracy									
	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	LightGBM	GAN
Ant	72.86	87.64	89.01	78.82	87.41	90.14	91.13	91.23	90.35	93.46
Camel	70.41	75.01	88.1	76.98	85.88	91.13	92.58	91.89	92.05	93.98
Ivy	75.34	87.11	89.87	75.11	86.49	89.3	91.08	92.11	91.78	91.77
jEdit	73.98	78.23	80.76	75.82	84.98	86.73	90.65	89.98	90.56	92.89
Lucene	77.95	86.07	87.43	78.12	88.67	92.05	92.47	91.45	93.41	94.02
Synapse	76.44	80.47	84.15	76.22	86.21	90.47	90.49	89.91	90.88	91.46
Tomcat	76.11	81.23	83.45	75.41	84.55	90.34	91.12	91.02	92.76	93.88
Velocity	75.89	80.47	82.32	75.89	86.32	86.23	87.66	88.56	87.22	91.57
Poi	76.21	84.76	84.94	77.11	86.79	87.54	86.21	87.11	88.56	89.42
Xalan	77.23	85.39	87.34	78.45	87.11	91.66	90.48	89.99	91.25	92.51
Average	**75.24**	**82.64**	**85.74**	**76.79**	**86.44**	**89.56**	**90.39**	**90.33**	**90.88**	**92.5**

Table 5: Performance metrics (AUC) for SDP models (Call Graph-based)

Projects	AUC									
	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	LightGBM	GAN
Ant	0.71	0.83	0.83	0.88	0.88	0.85	0.86	0.90	0.91	0.92
Camel	0.76	0.81	0.82	0.79	0.79	0.88	0.89	0.90	0.91	0.93
Ivy	0.75	0.80	0.84	0.85	0.85	0.85	0.85	0.91	0.90	0.91
jEdit	0.73	0.84	0.81	0.76	0.77	0.79	0.81	0.88	0.89	0.91
Lucene	0.76	0.81	0.80	0.78	0.81	0.88	0.89	0.90	0.93	0.93
Synapse	0.74	0.80	0.82	0.74	0.85	0.88	0.88	0.88	0.89	0.91
Tomcat	0.75	0.80	0.81	0.73	0.83	0.89	0.89	0.90	0.91	0.92
Velocity	0.74	0.81	0.81	0.73	0.84	0.84	0.86	0.87	0.86	0.89
Poi	0.76	0.82	0.83	0.75	0.84	0.85	0.84	0.86	0.87	0.88
Xalan	0.76	0.83	0.85	0.77	0.85	0.89	0.88	0.88	0.89	0.90
Average	**0.75**	**0.82**	**0.82**	**0.78**	**0.83**	**0.86**	**0.87**	**0.89**	**0.90**	**0.91**

While the other classifiers like MNB, LR, and KNN show respectable performance, GB, BAGD, and MLP classifiers generally demonstrate slightly lower accuracy and AUC scores. This is particularly noticeable in the Synapse and Tomcat projects, where the accuracy values are slightly lower for these classifiers. The average AUC across all projects for Call Graph-based models is 0.91, indicating that these models have a strong discriminatory power in predicting defects.

The Class Dependency Network-based SDP models (Table 6 and Table 7) show somewhat lower performance than the Call Graph-based models but still exhibit solid predictive capabilities. The highest accuracy is achieved with LightGBM (89.99% for the Lucene project) and GAN (88.09% for the Xalan project). On average, the accuracy for Class Dependency Network-based models is 74.17%, and the average AUC is 0.79, indicating a relatively good but less optimal performance when compared to the Call Graph-based models.

Table 6: Performance metrics (Accuracy) for SDP models (Class Dependency Network-based)

Projects	Accuracy									
	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	LightGBM	GAN
Ant	71.77	83.69	84.05	76.12	80.98	83.17	85.2	86.90	86.45	89.65
Camel	70.01	87.91	75.7	69.58	84.68	82.18	70.13	70.87	71.88	73.69
Ivy	76.56	86.99	82.57	74.45	82.49	89.76	89.08	89.34	88.71	90.55
jEdit	74.33	85.11	80.72	76.62	81.96	87.73	86.12	87.41	86.88	87.12
Lucene	74.85	82.02	83.44	74.19	85.01	88.98	87.91	88.32	89.11	89.99
Synapse	75.02	79.67	81.67	74.11	83.07	87.31	85.44	84.21	87.92	88.31
Tomcat	75.82	78.81	80.24	72.68	80.56	85.02	84.67	85.27	86.44	86.91
Velocity	73.45	79.01	81.45	73.31	83.26	81.78	82.01	82.78	83.55	83.12
Poi	73.89	82.33	80.67	74.67	84.05	82.37	80.47	81.09	82.47	83.88
Xalan	76.02	83.71	85.03	74.69	85.44	87.55	85.67	86.21	87.45	88.09
Average	**74.17**	**82.93**	**81.55**	**74.04**	**83.15**	**85.59**	**83.67**	**84.24**	**85.09**	**86.13**

Table 7: Performance metrics (AUC) for SDP models (Class Dependency Network-based)

Projects	AUC									
	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	LightGBM	GAN
Ant	0.70	0.80	0.81	0.72	0.76	0.79	0.78	0.80	0.80	0.84
Camel	0.74	0.82	0.81	0.70	0.72	0.77	0.66	0.69	0.70	0.72
Ivy	0.74	0.80	0.84	0.75	0.80	0.80	0.78	0.87	0.85	0.89
jEdit	0.73	0.80	0.78	0.73	0.77	0.79	0.76	0.85	0.84	0.84
Lucene	0.74	0.79	0.76	0.71	0.80	0.78	0.76	0.86	0.86	0.86
Synapse	0.72	0.76	0.79	0.72	0.82	0.84	0.84	0.83	0.85	0.86
Tomcat	0.73	0.75	0.78	0.70	0.78	0.83	0.83	0.84	0.84	0.85
Velocity	0.72	0.74	0.79	0.71	0.80	0.80	0.80	0.80	0.81	0.81
Poi	0.72	0.80	0.78	0.72	0.83	0.81	0.79	0.79	0.80	0.81
Xalan	0.71	0.80	0.83	0.72	0.83	0.85	0.84	0.84	0.85	0.85
Average	**0.73**	**0.79**	**0.8**	**0.72**	**0.79**	**0.81**	**0.78**	**0.82**	**0.82**	**0.83**

Notably, classifiers such as MNB, KNN, and MLP seem to perform more consistently, with MNB showing a notable accuracy of 76.56% for the Ivy project, which is slightly higher than in the Call Graph-based models. However, GAN and LightGBM still emerge as the most effective models for defect prediction in these networks, with higher AUC scores (0.85 for Lucene). The Traditional Software Metrics-based SDP models (Table-8) exhibit comparatively lower performance in terms of both accuracy and AUC when compared to both the Call Graph-based and Class Dependency Network-based models. While LightGBM still maintains a high level of accuracy (85.73% on average), other classifiers such as MNB and LR demonstrate a slight decrease in performance. For instance, MNB achieves an average accuracy of 72.74%, with LR following at 76.1%.

Table 8: Performance metrics for SDP models (software metrics-based)

Projects	Accuracy										AUC									
	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	LightGBM	GAN	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	LightGBM	GAN
Ant	70.89	79.69	84.05	73.12	79.98	80.03	82.72	83.65	84.91	86.02	0.70	0.77	0.83	0.72	0.79	0.79	0.80	0.82	0.83	0.82
Camel	70.01	72.91	75.7	69.58	78.68	70.13	82.18	84.22	83.98	85.97	0.74	0.82	0.81	0.70	0.72	0.77	0.79	0.83	0.82	0.82
Ivy	71.34	84.11	80.87	75.11	75.49	80.3	86.08	85.26	87.07	88.87	0.74	0.80	0.84	0.75	0.80	0.80	0.85	0.83	0.85	0.85
jEdit	74.33	80.11	75.72	76.62	80.96	76.73	80.12	81.84	82.15	84.02	0.73	0.80	0.78	0.73	0.77	0.79	0.80	0.80	0.80	0.83
Lucene	74.85	82.02	82.44	74.19	79.01	77.98	80.91	81.68	81.88	83.41	0.74	0.81	0.80	0.73	0.77	0.78	0.77	080	0.79	0.81
Synapse	72.87	71.54	72.21	73.67	76.78	75.45	80.56	81.93	83.14	84.22	0.70	0.70	0.72	0.71	0.74	0.74	0.79	0.80	0.82	0.80
Tomcat	73.22	72.34	74.56	74.87	78.13	78.23	81.45	81.79	82.49	85.31	0.71	0.71	0.72	0.72	0.76	0.76	0.80	0.80	0.81	0.80
Velocity	72.98	72.33	74.89	75.12	78.76	77.14	84.96	83.82	85.33	87.68	0.70	0.71	0.73	0.72	0.76	0.75	0.82	0.82	0.84	0.82
Poi	74.61	73.11	75.46	75.51	76.32	77.36	83.21	83.05	84.51	86.43	0.72	0.72	0.72	0.72	0.74	0.75	0.81	0.82	0.83	0.82
Xalan	72.34	72.87	73.57	76.23	78.46	79.98	82.44	82.62	83.77	85.33	0.70	0.70	0.71	0.74	0.77	0.76	0.80	0.81	0.81	0.80
Average	**72.74**	**76.1**	**76.95**	**74.4**	**78.26**	**77.33**	**82.46**	**82.99**	**83.92**	**85.73**	**0.72**	**0.75**	**0.77**	**0.72**	**0.76**	**0.77**	**0.8**	**0.81**	**0.82**	**0.82**

A significant takeaway from these results is that, despite the lower performance of the traditional software metrics-based models, they still offer reasonable defect prediction capabilities, with BAGD and KNN classifiers providing a decent balance between accuracy and AUC. However, these models do not reach the level of precision offered by the network-based models.

The Friedman test results (Table 9) highlight the statistical significance of the differences observed across the classifiers used in Call Graph-based and Class Dependency Network-based SDP models. The test values suggest that LightGBM and GAN consistently show significant performance improvements over other classifiers in both accuracy and AUC, especially in the Call Graph-based models, which consistently score higher than their Class Dependency Network counterparts.

Table 9: Results of statical test (Friedman Test)

Call Graph-based SDP models										
	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	Light GBM	GAN
Accuracy	10.04	8.78	7.31	9.64	8.71	6.68	6.15	5.01	4.67	3.31
AUC	9.78	6.21	5.29	8.02	7.54	4.58	4.27	3.99	3.15	2.91
Class Dependency Network-based SDP models										
Accuracy	9.41	8.33	6.34	9.11	7.34	5.68	5.65	5.18	4.88	4.11
AUC	8.12	6.01	5.78	7.02	4.98	4.58	4.17	4.11	3.96	3.21

In summary, the Call Graph-based SDP models consistently outperform the Class Dependency Network-based and Traditional Software Metrics-based models in terms of accuracy and AUC, with Light and GAN emerging as the most effective classifiers. However, the Class Dependency Network-based models still show promise, especially when utilizing Light and GAN. Traditional software metrics-based models, while performing lower on average, provide a viable alternative for projects where network representations might be unavailable or computational resources are limited.

The Friedman test results confirm the statistical significance of the superior performance of the network-based models, particularly for Call Graphs, validating their robustness and reliability in defect prediction tasks.

This study compares the performance of various classifiers across Call Graph-based and Class Dependency Network-based SDP models, using statistical analysis from the Friedman test (Table 9). The results reveal that the GAN classifier outperforms all other models in both accuracy and AUC, securing the lowest mean ranks for these evaluation parameters across both types of graphical representations. This establishes GAN as the best classifier for identifying software defects. Light follows as the second-best classifier, demonstrating strong performance in defect prediction as well. Both GAN and Light provide accurate defect predictions, but GAN's ability to handle imbalanced datasets by generating synthetic data gives it an edge in overall performance. The study also highlights a notable improvement in the performance of network-based models (using call graphs) when compared to traditional models based on software metrics, as shown in Figure 2.

The primary contribution of this study is the use of network metrics obtained by deriving from call graphs to predict software defects and analyze the associated costs on the basis of testing effectiveness. Unlike previous research, which focused mainly on class dependency networks that capture high-level relationships but neglect specific class dependencies, this study emphasizes a more detailed, granular approach using call graph metrics. It also expands on earlier studies by considering testing efficiency across different phases, which was previously overlooked.

Comparative Evaluation and Discussion:

The study provides a comparative analysis with earlier work, such as that of Gong et al. [14] and Biçer et al. [19], and demonstrates the advantages of using call graph-based network metrics for defect prediction. The cross-validation accuracy and AUC results validate the effectiveness of call graph-based models for software defect detection, establishing their potential for more accurate and cost-effective prediction.

Network Used: Gong et al. focused on Class Dependency Networks, which represent the high-level dependencies between software classes. Biçer et al. used Developer Communication Networks, focusing on relationships between developers rather than the software's internal structure. The present work employs both Call Graphs and Class Dependency Networks, which provide a more detailed and granular view of software dependencies at both the method-level and class-level. This approach enables more accurate defect prediction by leveraging structural insights from the code itself.

Dataset: Gong et al. worked with a diverse set of projects, including Groovy, HBase, ActiveMQ, Camel, Hive, JRuby, Derby, Wicket, and Lucene. Biçer et al. used IBM Rational Team Concert and Drupal data, which reflect developer interactions and communication patterns. The current study uses a narrower set of projects: Ant, Camel, Ivy, jEdit, and Lucene, allowing for more focused analysis within specific software domains and their respective defect prediction models.

Machine Learning Algorithms: Gong et al. employed Random Forest (RF) and Naive Bayes, which are traditional classifiers but with limited variety in handling complex defect prediction tasks. Biçer et al. used Naive Bayes, a simpler approach in comparison to the modern techniques used in the present work. The present study takes a more comprehensive approach by using a wide range of classifiers, including MNB, LR, RF, KNN, GB, BAGD, MLP, Light, LSTM, and GAN. The inclusion of advanced techniques like Light, LSTM, and GAN offers a broader and more robust comparison, allowing for better performance in defect prediction due to the use of newer machine learning methods capable of handling complex data patterns.

Model Evaluation: Gong et al. used statistical tests like the Friedman Test, Wilcoxon-signed rank Test, and the Nemenyi Test for model evaluation, which provides a detailed statistical comparison of models. Biçer et al. evaluated models by looking at the probability of false alarms and detection, which focuses more on the balance between precision and recall but doesn't fully evaluate overall model accuracy. The present work focuses on prediction accuracy and AUC, complemented by the Friedman Test for a thorough statistical analysis of classifier performance. This allows for a more comprehensive understanding of model effectiveness in defect prediction.

SDP Context: Gong et al. conducted their study in within-project, cross-version, and cross-project contexts, which allowed for a broader evaluation of the models across different projects and versions. Biçer et al. only focused on the within-project context, looking at defect prediction within the same project over different versions. The current study also evaluates models in a within-project context, but it goes further by incorporating network metrics from both Call Graphs and Class Dependency Networks, providing a more refined analysis of software defects within the same project.

Key Observations: The present work extends previous research by combining the power of Call Graphs and Class Dependency Networks, providing a more comprehensive understanding of software defects at multiple levels. It leverages a wider array of modern machine learning classifiers, surpassing the more traditional approaches used in past studies.
The inclusion of detailed cost evaluation, considering the testing effectiveness across different stages of the software development process, marks a significant advancement in understanding the practical implications of SDP models. The comparative analysis highlights the improvements in defect prediction accuracy and AUC, demonstrating the advantages of using network-based metrics derived from Call Graphs, which have been shown to outperform previous approaches that relied solely on Class Dependency Networks.

5. Conclusion and Future Work

The proposed SDP approach that leverages software networks—specifically, call graphs—while incorporating cost considerations into defect prediction. The SDP model developed in this study performs better than the traditional SDP approaches that rely only on software metrics, as well as the models utilizing network metrics derived from alternate network representations such as class dependency networks. By training ten different classifiers and extracting network metrics from call graphs, we provide a comprehensive assessment of SDP performance and compare it with models based on class dependency networks. Our findings demonstrate that call graphs serve as an excellent graphical representation of software systems. The model was tested across ten Java projects, showing AUC improvements ranging from 2.9% to 8.94% on most datasets, attributed to the model's ability to capture complex relationships among software components—an essential factor in effective SDP. Among the classifiers, the GAN achieved the highest success, with an AUC of 0.91 and accuracy of 92.5%, largely due to its capacity to handle imbalanced datasets.

For future work, the model can be enhanced by incorporating data balancing and feature selection techniques to remove redundant or irrelevant features, potentially improving prediction accuracy. Moreover, extending the assessment to a wide variety of defect datasets across various projects would provide deeper insights into the model's generalizability and robustness, validating its performance in diverse software development contexts.

Conflict of Interest: All authors declare that he/she has no conflict of interest.

REFERENCES

1. Raamesh L, Jothi S, Radhika S. Enhancing software reliability and fault detection using hybrid brainstorm optimization-based LSTM model. IETE J Res. 2023;69(12):8789–803. https://doi.org/10.1080/03772063.2022.2069603
2. Jureczko M, Madeyski L. Towards identifying software project clusters with regard to defect prediction. In: Proc 6th Int Conf Predictive Models Softw Eng. 2010. p. 1–10.
3. Khleel NAA, Nehéz K. Software defect prediction using a bidirectional LSTM network combined with oversampling techniques. Clust Comput. 2024;27(3):3615–38. https://doi.org/10.1007/s10586-023-04170-z
4. Gong L, Rajbahadur GK, Hassan AE, Jiang S. Revisiting the impact of dependency network metrics on software defect prediction. IEEE Trans Softw Eng. 2021;48(12):5030–49. https://doi.org/10.1109/TSE.2021.3131950
5. Ulan M, Löwe W, Ericsson M, Wingkvist A. Weighted software metrics aggregation and its application to defect prediction. Empir Softw Eng. 2021;26(5):86. https://doi.org/10.1007/s10664-021-09984-2
6. Arar ÖF, Ayan K. A feature dependent Naive Bayes approach and its application to the software defect prediction problem. Appl Soft Comput. 2017;59:197–209. https://doi.org/10.1016/j.asoc.2017.05.043
7. Halstead MH. Elements of Software Science. New York: Elsevier Science Inc.; 1977.
8. Chi J, Qu Y, Zheng Q, Yang Z, Jin W, Cui D, et al. Test case prioritization based on method call sequences. In: 2018 IEEE 42nd Annu Comput Softw Appl Conf (COMPSAC). 2018;1:251–6.
9. —
10. Alqarni A, Aljamaan H. Leveraging Ensemble Learning with Generative Adversarial Networks for Imbalanced Software Defects Prediction. Appl Sci. 2023;13(24):13319.
11. Singh Y, Kaur A, Malhotra R. Prediction of fault-prone software modules using statistical and machine learning methods. Int J Comput Appl. 2010;1(22):8–15.
12. Ponnala R, Reddy CRK. Ensemble model for software defect prediction using method level features of spring framework open source Java Project for E-Commerce. J Data Acquis Process. 2023;38(1):1645.
13. Antal G, Tóth Z, Hegedűs P, Ferenc R. Enhanced bug prediction in JavaScript programs with hybrid call-graph based invocation metrics. Technol. 2020;9(1):3. https://doi.org/10.3390/technologies9010003
14. Gong L, Rajbahadur GK, Hassan AE, Jiang S. Revisiting the impact of dependency network metrics on software defect prediction. IEEE Trans Softw Eng. 2021;48(12):5030–49. https://doi.org/10.1109/TSE.2021.3131950

15. Setia S, Ravulakollu KK, Verma K, Garg S, Mishra SK, Sharan B. Software Defect Prediction using Machine Learning. In: 2024 11th Int Conf Comput Sustain Glob Dev (INDIACom). IEEE; 2024. p. 560–6.
16. Ma W, Chen L, Yang Y, Zhou Y, Xu B. Empirical analysis of network measures for effort-aware fault-proneness prediction. Inf Softw Technol. 2016;69:50–70. https://doi.org/10.1016/j.infsof.2015.09.001
17. Zhou C, He P, Zeng C, Ma J. Software defect prediction with semantic and structural information of codes based on graph neural networks. Inf Softw Technol. 2022;152:107057. https://doi.org/10.1016/j.infsof.2022.107057
18. Sawadpong P, Allen EB. Software defect prediction using exception handling call graphs: A case study. In: 2016 IEEE 17th Int Symp High Assur Syst Eng (HASE). 2016. p. 55–62.
19. Biçer S, Bener AB, Çağlayan B. Defect prediction using social network analysis on issue repositories. In: Proc 2011 Int Conf Softw Syst Process. 2011. p. 63–71.
20. Xu J, Ai J, Shi T. Software Defect Prediction for Specific Defect Types based on Augmented Code Graph Representation. In: 2021 8th Int Conf Dependable Syst Their Appl (DSA). IEEE; 2021. p. 669–78.
21. Wu W, Wang S, Liu B, Shao Y, Xie W. A novel software defect prediction approach via weighted classification based on association rule mining. Eng Appl Artif Intell. 2024;129:107622. https://doi.org/10.1016/j.engappai.2023.107622
22. Li Z, Niu J, Jing XY. Software defect prediction: future directions and challenges. Autom Softw Eng. 2024;31(1):19. https://doi.org/10.1007/s10515-024-00424-1
23. Boucher A, Badri M. Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. Inf Softw Technol. 2018;96:38–67. https://doi.org/10.1016/j.infsof.2017.11.005
24. Zimmermann T, Nagappan N. Predicting defects using network analysis on dependency graphs. In: Proc 30th Int Conf Softw Eng. 2008. p. 531–40.
25. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. IEEE Trans Softw Eng. 1994;20(6):476–93.
26. Wolf T, Schroter A, Damian D, Nguyen T. Predicting build failures using social network analysis on developer communication. In: 2009 IEEE 31st Int Conf Softw Eng. IEEE; 2009. p. 1–11.
27. Nguyen TH, Adams B, Hassan AE. Studying the impact of dependency network measures on software quality. In: 2010 IEEE Int Conf Softw Maint. 2010. p. 1–10.
28. Kumar PS, Venkatesan R. Improving Software Defect Prediction using Generative Adversarial Networks. Int J Sci Eng Appl. 2020;9:117–20.
29. Rebro DA, Chren S, Rossi B. Source Code Metrics for Software Defects Prediction. In: Proc 38th ACM/SIGAPP Symp Appl Comput. 2023. p. 1469–72.
30. Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R. Toward a smell-aware bug prediction model. IEEE Trans Softw Eng. 2017;45(2):194–218. https://doi.org/10.1109/TSE.2017.2770122
31. Sharma D, Chandra P. An empirical analysis of software fault proneness using factor analysis with regression. Multimed Tools Appl. 2024;83(17):52535–91.
32. McCabe TJ. A complexity measure. IEEE Trans Softw Eng. 1976;(4):308–20.
33. Deng J, Lu L, Qiu S. Software defect prediction via LSTM. IET Softw. 2020;14(4):443–50. https://doi.org/10.1049/iet-sen.2019.0149
34. Gupta S, Gupta A. A set of measures designed to identify overlapped instances in software defect prediction. Comput. 2017;99:889–914. https://doi.org/10.1007/s00607-016-0538-1
35. Qu Y, Zheng Q, Chi J, Jin Y, He A, Cui D, et al. Using k-core decomposition on class dependency networks to improve bug prediction model's practical performance. IEEE Trans Softw Eng. 2019;47(2):348–66. https://doi.org/10.1109/TSE.2019.2892959

About the Authors

Sweta Mehta is an Assistant Professor in the Department of Computer Science and Engineering at Sarala Birla University. She holds an M.Tech degree from BIT Mesra, Ranchi, and is currently pursuing her Ph.D. Her academic interests include software engineering, machine learning, and emerging computing technologies.