Optimizing Cost Efficiency in Software Defect Prediction Through Network Representation Methods

Sweta Mehta[1], Pankaj K. Goswami[1], K.Sridhar Patnaik[2],

[1] Department of CSE, Sarala Birla University, Ranchi, India
[2] Department of CSE, Birla Institute of Technology, Mesra, Ranchi, India

sweta.mehta949@sbu.ac.in, pankaj.goswami@sbu.ac.in, kspatnaik@bitmesra.ac.in,

Corresponding Author: Sweta Mehta, Sarala Birla University, Ranchi, India
sweta.mehta949@sbu.ac.in

Abstract

The significance of software defect prediction (SDP) has been well established owing to its usefulness in preventing potential defects in software at the earliest possible phase within its development cycle. Research works in SDP utilizing traditional metrics related to code complexity and coupling does not have the capability to capture the interrelationships and interactions that are a common characteristic in big software systems. A better modelling of the underlying structural relationships within the software is required to design an efficient and accurate SDP model. Network based graphical representations such as call graphs and class dependency networks have the potential to capture the intracies of the dependencies and the hidden patterns among those dependencies. Call graphs map the function level interactions in the form of caller-callee relationship of the function calls within the software while the class dependency network map the module level dependencies within the software. This study aims to evaluate call graphs and class dependency networks for a cost effective and highly accurate framework of software defect prediction. The evaluation comprises of ten machine learning classifiers utilizing the call graphs and class dependency networks of ten real software projects based on Java. The findings indicate the superiority of call graphs compared to class dependency networks as the SDP model based on improvement in AUC ranging from 2.9% to 8.94% for majority of the datasets owing to their ability to better capture the intricate software component relationships which is a critical aspect in SDP. Generative Adversarial Network proved to be the most successful classifier, among the evaluated classifiers with the AUC of 0.91 and an accuracy of 92.5%.

Keywords: Software Defect Prediction, Generative Adversarial Networks, Network Representation, Class Dependency Networks, Call Graphs, Cost Analysis

1. Introduction

Software testing is a widely acknowledged resource-intensive phase among the various phases of the software development life cycle (SDLC). It serves as the crucial stage in SDLC as it ensures the quality of the software and is thus regarded as the most expensive phase. This creates a need to enhance testing efficiency so that the testing resources are utilized effectively [1]. Software defect prediction serves as one such mechanism that reduces the occurrence of defects by identifying at an early stage of SDLC the modules comprising the software that have an increased probability of containing defects. This prediction of defect-prone software modules not only increases testing efficiency but also enables module-specific targetted testing thus reducing the effort of the testing teams and other resources involved in various stages of SDLC. Software defect prediction models also greatly the developers to be more aware when developing or modifying the code in further maintenance phases. This module-specific targetted testing approach helps to deliver quality software with reduced cost

overhead of the resources involved [5, 14]. As the defect-prone modules are detected at an initial stage, it significantly reduces the defect count at the testing phase along with the introduction of new defects in the maintenance phase.

Conventional SDP techniques have widely adopted software metrics as the basis for developing the SDP models by representing the software in the form of complexity and code metrics. These metrics have helped map the code coupling and complexity details of the software components mainly at the class level. With the inadvertently increasing complexity of the software, the software metrics cannot keep up and accurately represent the dependencies and intricate relationships between the software components [2, 4]. Thus the SDP models developed utilizing the software metrics do not seem to perform well over the current software and draw attention towards the need for SDP models that can provide accurate prediction results at a relatively low cost. This has led to the utilization of advanced techniques which include various code representation techniques that capture static and dynamic interactions among the code along with advanced machine learning classifiers to more efficiently predict the defect-prone modules. The development of robust SDP models will significantly enhance the reliability of the software systems [7].

Among the advanced techniques, some studies have used network metrics based on Social Network Analysis to provide insight into the architectural structure and relationships within the codebase, enhancing the SDP performance of the models. Within the studies using the graph-based software representations, studies have mainly used class dependency networks. The class dependency networks provide a high-level view of the component relationships. However, this focus of class dependency network leaves other potentially valuable graphical representations that can enhance SDP underexplored. Recognizing this research gap, our study focuses on investigating the role of alternative network representations, specifically call graphs, in enhancing defect prediction accuracy and the cost effectiveness of the SDP model.

2. Existing Research

Software defect prediction has gained popularity in recent years in software engineering. Various researchers have developed SDP models by combining a variety of techniques. These mainly include combining software metrics such as static code metrics, and code churn metrics along with network representations such as software module networks. The static code metrics widely used in many studies only take into consideration the structural features of the software's source code [3, 15]. Singh et al. [18] used the NASA AR1 dataset, which includes a variety of software measures to evaluate various regression and machine learning techniques for SDP. Their decision tree algorithm-based model fared better than models based on regression and other machine learning techniques, making it more interpretable for defect prediction and especially excellent at capturing intricate relationships between software indicators. Zimmermann et al. [23] identified the role of "change bursts" and highlighted its better performance over conventional metrics like complexity of code and code churn in detecting faults. Using data complexity metrics, Gupta et al. [16] examined 54 software projects through a comprehensive defect dataset of 327 datasets to find overlaps. Classifiers that were trained on non-overlapping datasets performed better on test data that had overlaps. The significance of dataset quality and effort-sensitive evaluation is highlighted by the fact that models also performed better when evaluation measures took defect discovery effort into account.

Applied to a range of problems, Social Network Analysis (SNA) has also become increasingly popular in software engineering. Wolf et al. [21], for instance, used data from the RTC release 1 repository and SNA to examine networks such as those of developers communications for forecasting failures in the builds. Instead of concentrating on build failures, our research adopts a new strategy by predicting modules that are prone to defects. Researchers have attempted to develop SDP models with alternate methods including SNA. Based on the study of Ma et al. [25] compared with the performance of software code metrics, SNA provided promising

results for the SDP study on within-project and cross-version scenarios. Comparably, Nguyen et al. [22] showed through experimental evaluation that the SDP model's performance is enhanced by integrating SNA measurements with code metrics, especially when applied to a particular project. The results obtained highlight the positive effects of using a combination of metrics, data science, and machine learning techniques in SDP. Boucher et al. [3] conducted an extensive analysis of a variety of software metrics establishing a connection between the defect-prone modules and their respective software metrics. The obtained results highlight the potential of utilizing a combination of metrics to improve the SDP model's performance. This study also examined various methods for establishing the value limts for the software's traditional metrics with results indicating the Alves ranking and ROC curve as the acceptable parameters. Ulan et al. [24] presented an unsupervised learning-based automated method using weighted aggregate metrics. This approach combined the probability theory concepts to determine the weights and scores of metrics. This study focused mainly on the software and change metrics. The evaluation's applicability to a broader variety of software applications was also called into doubt due to its restriction to a small number of object-oriented measures.

The previous studies exploring various techniques for SDP have largely utilized traditional software metrics which are useful in SDP to a certain extent but seem to overlook the structural relationships between the modules in the software [33]. Recent research works have started to incorporate graph or network-based representations of software to capture the dependencies and interactions between software modules. A major work in this area was presented by Qu et al. [30], wherein a Class Dependency Network was utilized for k-core decomposition to rank the defective classes to improve the SDP performance. Zhou et al. [31] utilized the class dependency network of the software to obtain the semantic and structural dependency data about the software by using Abstract Syntax Trees and Network Embedding techniques. In addition to these works, the study by Antal et al. [32] proposed an SDP model that focused on function-level hybrid metrics. Although Their work used this approach to validate it for JavaScript Projects, such research works need to be validated for other mainstream programming languages.

Raamesh et al. [17] integrated optimization algorithms to propose a hybrid LSTM based model for defect detection and correction. The defect datasets of Firefox and Bugzilla are utilized to evaluate the performance of the proposed framework based on mean squared error evaluation parameter. Ponnala & Reddy [26] utilized Random Forest, Support vector machine, and Light Gradient Boosting machine to propose an ensemble approach considering method-level information for developing a defect prediction model. Their approach mainly focused on the method caller-callee relationship, the length of the methods, and their complexity, thus achieving a ROC value of 0.853. Their work only utilized a single Java project called Broadleaf Commerce therefore it is necessary to evaluate its effectiveness across a wide range of projects. The research conducted by Kumar & Venkatesan [12] highlighted the benefits of using GAN for software defect prediction however their research work utilized very few projects therefore raising the need for its validation over a wide range of projects. Alqarni & Aljamaan [14] used an ensemble model comprising AdaBoost for SDP in combination with GAN to generate a synthetic dataset to build a stable SDP model that is not based on a skewed defect dataset. Their evaluation results indicated the benefits of GAN-based models over traditional SDP models. However in these studies discussed above the defect dataset used was software metrics which provides minimal insights into complex software dependencies that form the root cause of software defects, in turn resulting in a less effective software defect prediction model and raising the need for the development of SDP models that are more effective in capturing the complex interactions among the software modules.

3. Methodology

3.1 Metrics for Software

Software metrics represent the quantified form of the various aspects of the software. This section describes the types of software metrics used by the SDP studies. This study is focused around the following categories of software metrics: traditional software metrics and network metrics based on social network analysis.

3.1.1 Conventional Software Metrics

Software systems characteristics, features, dependencies, and complexities are quantitatively measured using software metrics. These are also regarded as the traditional software metrics in software engineering. The most initial and important types of metrics were established by Halstead [10] and McCabe [34]. These metrics focus on the basic understanding of the software. Further, with the development of software systems pertaining to object-oriented programming paradigms, CK metrics [19] were designed to focus largely on object-oriented concepts to better model the characteristics of software. These metrics generally comprise complexity metrics such as cyclomatic complexity, code churn metrics, and size related metrics. The size metrics mainly deal with estimating the size of the software based on certain parameters such as lines of code, on the other hand, the complexity metrics are associated with determining the complexity levels of the source code depending on the number of linearly independent associations between the software components. Code churn metrics focus on the software components or modules that are constantly going under some or the other change during the evolution of the software or the later phases such as during maintenance. Object-oriented based software metrics take into consideration object-oriented core design frameworks, mainly the coupling, and cohesion between the software modules. It also maps the inheritance relationships between the components to measure the design qualities of the software that may impact the defect proneness of the software components.

3.1.2 Network Metrics

Networks provide a graphical representation of concepts that are difficult to comprehend. By visualizing software as a network of connected software components it becomes easy to map the complex interactions among the software components. The networks provide structural knowledge with the help of metrics known as network metrics which map the network structure and relation between the nodes in a quantative measure making it easier to use for the underlying tasks, which is Software defect prediction concerning this study. These metrics have been derived from studies based on Social Network Analysis (SNA) [21]. In this study, the network metrics described in Table 1 are obtained from graphical representations such as call graphs and class dependency networks.

Different measures of the network metrics quantify how the nodes represented by various software components in the network representations interact with each other. Network Measures such as degree centrality measures a node's influence on the nodes in its direct connection. The software modules having a high degree of centrality represent the modules essential and critical to the functioning of the software. Betweenness centrality identifies the connecting modules in the software system that regulate the information flow and may be crucial to identifying the defect-prone modules. The software components' interconnectivity and redundancy can be mapped in quantitative measure by the metric clustering coefficient which measures how much the software components tend to cluster together leading to increased complexity of the software. Such network measures help in the identification of the modules prone to defects within the software by providing a deeper insight into the software system's structure and behavior.

Table 1: Description of network metrics

S. No.	Metric	Definition
1.	Pairs	Total count of distinct node pairs
2.	Ties	edge count represents the total count of directed ties

3.	Size	Count of nodes to which the ego is immediately linked
4.	Density	The proportion of potential ties available currently
5.	nWeakComp	Total weak components / size
6.	ReachEfficiency	2StepReach / size
7.	2StepReach	The proportion of nodes that are present after two steps
8.	EgoBetween	The proportion of routes with the shortest distance between neighbors that go through ego.
9.	Broker	Total pair of nodes that aren't linked directly.
10	nBroker	Broker / size
11.	Betweenness	Determines the count of shortest paths existing between all the other entities
12.	Reachability	Nodes accessible from a specific node
13.	Efficiency	Effective size of a network / network's total size
14.	Hierarchy	The distribution of the constraint metrics all over the neighbours
15.	Degree	Total count of nodes next to a specific node
16.	Closeness	The total length of all shortest routes from a specific node to all the remaining nodes
17.	Constraint	Measures the degree of a node's constraints
18.	Power	Specifies the number of links a node has in its neighborhood
19.	Eigenvector	Assigns the nodes of the dependency graph with relative scores

3.2 Network Representations

The network representations map the various software components into nodes and the interactions between them as the connections between the nodes. These network representations have effectively helped in the efficient evolution, optimization, and maintenance of the software system by identifying the structural and inter-component relationships in the software's source code. This section discusses the details of the two network representation techniques used in this study: Call Graphs and Class Dependency Networks.

3.2.1 Call Graphs

Call graphs are a network representation technique that represents the software as a network by mapping the function calls between the software modules [6, 27]. In call graphs, the nodes represent methods or functions present in the source code and the edges represent the control flow between the functions. The function level caller-callee relationship is indicated through directed edges from the caller function or the invoking function to the callee function or the invoked function. This network of function calls can be mapped to be viewed as the dependency between classes or other software components based on the function calls between them. Call graphs can be static or dynamic depending on when it was generated. Static call graphs are generated using the source code before the code is executed just by examining the functions and their relationships as described in the code. It provides a thorough comprehensive understanding of the code. The dynamic call graphs are generated simultaneously during the execution of the code providing insights into the behavior of the software under specific circumstances providing detailed analysis on the runtime interactions.

This study utilizes static call graphs representation which helps to identify the defect-prone modules owing to its in-depth representation of dependencies. This study is based on the understanding that more the number of function calls within the classes, the higher is chance of defects being introduced in those classes during the development and maintenance phases owing to the highly complex control and data flow. Thus it helps to

design SDP models that focus on more fine-grained dependencies that can help identify possible system bottlenecks and areas of the code that can be further optimized to avoid future occurrence of defects [28].

3.2.2 Class Dependency Networks

Class dependency network is a widely used graphical representation of the software that mainly maps how the classes in an object oriented software system are related to one another. The nodes in this network representation are either classes of interfaces and the edges represent the conncetion between them providing a representation of the dependencies between various parts of the system by focusing on the structural links between the software classes [29]. This representation is useful to visualize the softwares architecture and helps to understand the code modularity facilitating the understanding of code interactions, inheritance hierarchy, series of linked class dependencies. These also form the basis on which the connections in the class dependency network rely. Class dependency networks capture the structural complexity at a very high or abstract level based on the above-discussed parameters. Class dependency networks are used for software defect prediction because defects within a software module or component get propagated to all the dependent modules and components. Thus, defect prediction models have incorporated the dependency-related information from the class dependency networks for identifying the defect-prone modules and components.

3.3 Machine learning algorithms

This study aims to predict defect-prone modules, specifically classes in the software by categorising the software classes as either defect-prone or non-defect-prone. In order to complete this binary classification task, a variety of classifiers representing different categories is evaluated [13]. To diversify developed models various classifier categories ranging from simple classifiers to ensemble techniques and advanced variants of neural networks is used in this study. Among these categories are tree-based classifiers, which use decision trees to make predictions based on feature values; instance-based classifiers, which base their decisions on comparisons with training instances; probabilistic classifiers, which estimate probabilities for class membership; and linear classifiers, which divide classes using linear decision boundaries. Deep Learning Classifiers use neural networks to identify intricate patterns, while Ensemble Classifiers integrate several classifiers to increase prediction accuracy. As an example of the variety of approaches investigated to improve the precision of defect prediction in software systems, Table 2 presents the seven classifiers used in this investigation. The strengths of these classifiers for predicting software defects are well-balanced. Simplicity and interpretability are offered by multinomial Naive Bayes and logistic regression, particularly when working with small datasets. Generalisation is improved and unbalanced data is efficiently handled by Random Forest and Bagging + Decision Trees. K-Nearest Neighbours uses feature similarity to identify patterns without making parametric assumptions, whereas Gradient Boosting focusses on error correction to provide strong predictions for datasets that are not balanced [1]. The Multi-Layer Perceptron with two hidden layers is used in this study to capture subtle, non-linear interactions, which allows the model to respond to the complex nature of defect prediction. Long Short-Term Memory (LSTM), which is a type of Recurrent Neural Network, is included as an algorithm for developing the SDP model for its capability to model complex feature representations [35]. Considering the imbalance in the defect datasets due to low percentage of defective classes, the evaluation conducted in this study also includes the machine learning algorithms capable of handling such imbalance. This includes LightGBM and Generative Adversarial Network wherein the former is a variant of a boosting ensemble while the latter is a variant of Neural Network.

Table 2: Machine Learning algorithms used in this study.

S. No	Classifier	Notation
1	Multinomial Naive Bayes	MNB
2	Logistic Regression	LR
3	Random Forest	RF

4	K-Nearest Neighbors	KNN
5	Gradient Boosting	GB
6	Bagging + Decision Tree	BAGD
7	Multi Layer Perceptron (2 hidden layers)	MLP
8	Long Short-Term Memory	LSTM
9	Light Gradient Boosting Machine	LightGBM
10	Generative Adversarial Network	GAN

3.4 Dataset Description

The proposed framework is evaluated using software projects based on Java programming language. The projects and defect dataset are obtained from the PROMISE repository [11], which provides the details of these projects in the form of defect data obtained at the class level. Table 3 summarizes the details of these projects. It presents the total number of classes present along with the total number of defective classes followed by the percentage of defective classes. The defect data of every project contains the twenty class-level software metrics for each class followed by a column indicating whether the respective class contains defects or not. A specific version of a total of ten projects is selected based on varied percentages of defects ranging from 8.97% to 63.57%. This varied defect percentage brings the required diversity among the projects resulting in the development of a robust generalized SDP model.

Table 3 : Detailed description of the software projects

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

3.6 Cost Evaluation

The cost associated with using an SDP model needs to be evaluated to compute the cost-effectiveness of the developed model so that it can be widely used in practice in the industry. When a module predicted as defect-prone is non-defective or vice versa, the possible impact of this misclassification of the module on the effort required to test the modules needs to be considered by the SDP framework aiming to achieve cost-effectiveness. The incorrectly classified modules of the software drastically increase the effort required by the testing teams, owing to the nature of defects and also due to the propagation of the existing defects to the initially unaffected modules, thus increasing the financial implications. The defects that do not get detected at an early stage become the cause of many more defects that get introduced in the software, thus increasing the maintenance costs due to the increased complexity of fixing these defects. Thus correct identification of the defect and non-defect prone modules is very important but simultaneously, the misclassification cost of the SDP model needs to be considered for its effective utilization. To very nearly estimate the cost of fixing the misclassified

modules, the cost associated with fixing defects at different testing stages such as unit, integration, and system testing needs to be determined and then added up to compute the total cost involved in software defect resolution. A cost analysis framework taking into consideration the cost involved at various testing stages was proposed by Kumar et al. [9], which is centered around two factors: the first being the estimated cost of defect removal using a software defect prediction model (Ecost)and the second factor being the defect removal cost without using any SDP model (Tcost). The study introduced the concept of normalized cost (NEcost) based on these two factors Ecost and Tcost. Necost is the indicator of how well the SDP model reduces the testing cost. It is considered that if the NEcost values are less than 1, then the SDP model is considered cost-effective while if the NEcost value is more than 1 it indicates that using the SDP model is not cost-effective and the conventional testing methods are better to use in this case for effective utilization of resources. By this analysis, the stakeholders can determine the economic feasibility of using the SDP models in their projects.

3.6 Overview of experimental setup

Network representation of the source code models the relationships and interactions between the software components. This study utilizes two network representations: class dependency networks and call graphs to map the class-level dependencies at different levels of granularity and detail. Section 3.2 discusses the details of these two types of representations. Network metrics have been utilized in this study to gain insights from the distinct structural information obtained from the network. The study provides an in-depth comparison of the performance of network metrics in predicting defects in software compared to conventional metrics. In addition, the cost analysis is performed for both types of metrics to determine if using network metrics results in any improvement in the cost-effectiveness of the model.

The major steps in this study are initiated by obtaining the source code of the projects from their respective code bases, followed by obtaining the defect data of those projects from the well recognized PROMISE repository [11] in the software engineering domain. The static representations of the Call Graph and Class Dependency Network are produced by parsing the source code of each project using the Understand[1] tool thus resulting in two graphical representations for each project. The next stage involves capturing the features of the network representations by obtaining the network metrics from the call graphs and class dependency networks using the Ucinet[2] tool. Once the network metrics are obtained, the defect dataset to train the SDP model is designed by associating the class-level defect label information obtained from the PROMISE repository with the respective class-level network metrics. By correlating the network metrics of a class with the presence or absence of defects, the resulting dataset captures the structural relationships at different granularity levels each in the case of call graphs and class dependency networks.

[1] https://scitools.com/
[2] https://sites.google.com/site/ucinetsoftware/home

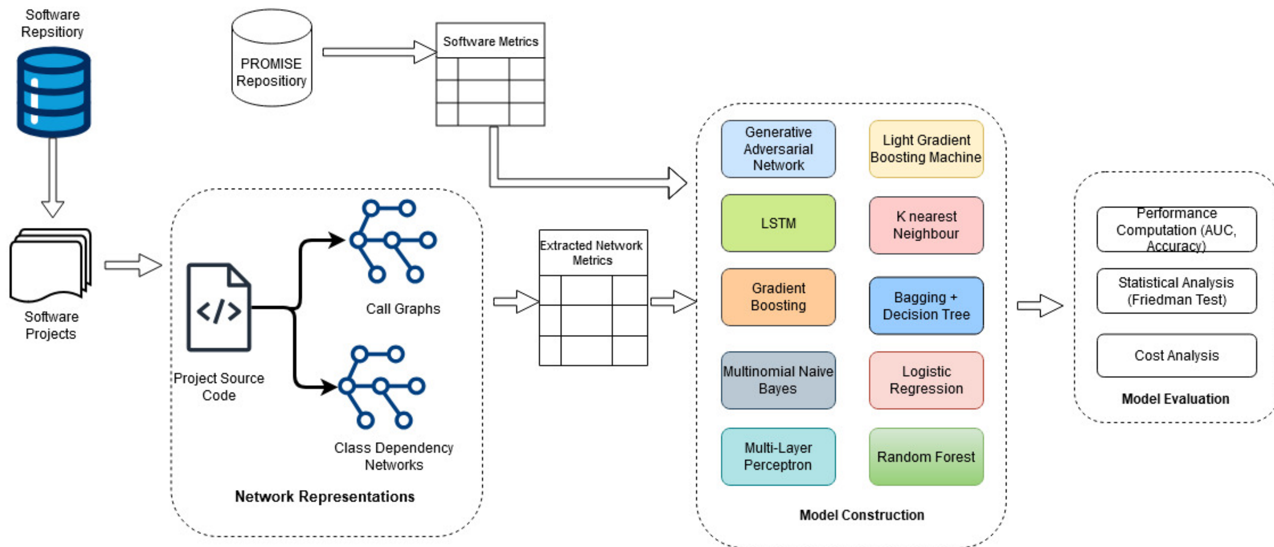

Fig. 1: Overview of the study design

Further, This dataset is used to first train the developed SDP model followed by its testing where the network metrics are used as the predictive features while the defect information is used as the target or output label. To perform a comparative analysis of the effect of network metrics based on call graphs and class dependency networks, SDP models are developed using both these types of networks and the performance is evaluated and comparison is drawn on the basis of evaluation metrics: Area Under the Curve (AUC) and accuracy. This evaluation is further strengthened by comparing the performance with the traditional software metrics. For the development of SDP models, seven different machine learning classifiers are utilized to identify the best-performing classifier within the SDP model setup making use of network metrics. To further validate the findings of the study, a statistical test, such as the Friedman test, is utilized to assess the effectiveness of the proposed SDP models and determine whether their differences are statistically significant or not. Lastly, the cost analysis of the models is performed by taking into consideration the low, medium, and high testing effectiveness of the testing teams across different testing phases of the software development. The insights gained from this analysis of cost help identify the top-performing SDP model taking into consideration the two major factors: defect prediction performance and cost-effectiveness, which helps to present a cost-effective, generalized robust SDP model. Figure 1 presents an overview of the major stages involved in developing the proposed SDP framework.

4. Experimental Results

Assessing the effects of various software graphical representations on the cost of identifying software defect-prone classes is the main goal of this study. The study's experimental findings are presented in this section. The performance metrics for the SDP models based on Call Graphs across different projects are shown in Table 4, particularly accuracy and AUC. Likewise, Table 5 offers the same metrics for the Class Dependency Network-based models. Furthermore, the models' performance metrics that were obtained from Software Metrics are displayed in Table 7.

Following a thorough examination of these findings, the following important conclusions are drawn:

- Call Graph-based SDP Models: Of the classifiers employed in this category, the Generative Adversarial Network proved to be the most successful, with the best AUC of 0.91 and an accuracy of 92.5%. Closely

following, the LightGBM classifier demonstrated excellent performance as well, with a 0.90 average AUC value.

- Class Dependency Network-based SDP Models: Among the two classifiers employed in this SDP model setup, the Generative Adversarial Network obtained a mean AUC value of 0.83, which is higher than the average AUC obtained for the rest of the classifiers. Also, the average accuracy obtained across all the projects is 86.13%.
- Comparative Analysis: In most datasets, models based on Call Graphs constantly performed better than those based on Class Dependency Networks when evaluating overall performance across various projects.

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

Projects	Accuracy										AUC									
	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	LightGBM	GAN	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	0.71	0.83	0.83	0.88	0.88	0.85	0.86	0.90	0.91	0.92
Camel	70.41	75.01	88.1	76.98	85.88	91.13	92.58	91.89	92.05	93.98	0.76	0.81	0.82	0.79	0.79	0.88	0.89	0.90	0.91	0.93
Ivy	75.34	87.11	89.87	75.11	86.49	89.3	91.08	92.11	91.78	91.77	0.75	0.80	0.84	0.85	0.85	0.85	0.85	0.91	0.90	0.91
jEdit	73.98	78.23	80.76	75.82	84.98	86.73	90.65	89.98	90.56	92.89	0.73	0.84	0.81	0.76	0.77	0.79	0.81	0.88	0.89	0.91
Lucene	77.95	86.07	87.43	78.12	88.67	92.05	92.47	91.45	93.41	94.02	0.76	0.81	0.80	0.78	0.81	0.88	0.89	0.90	0.93	0.93
Synapse	76.44	80.47	84.15	76.22	86.21	90.47	90.49	89.91	90.88	91.46	0.74	0.80	0.82	0.74	0.85	0.88	0.88	0.88	0.89	0.91
Tomcat	76.11	81.23	83.45	75.41	84.55	90.34	91.12	91.02	92.76	93.88	0.75	0.80	0.81	0.73	0.83	0.89	0.89	0.90	0.91	0.92
Velocity	75.89	80.47	82.32	75.89	86.32	86.23	87.66	88.56	87.22	91.57	0.74	0.81	0.81	0.73	0.84	0.84	0.86	0.87	0.86	0.89
Poi	76.21	84.76	84.94	77.11	86.79	87.54	86.21	87.11	88.56	89.42	0.76	0.82	0.83	0.75	0.84	0.85	0.84	0.86	0.87	0.88
Xalan	77.23	85.39	87.34	78.45	87.11	91.66	90.48	89.99	91.25	92.51	0.76	0.83	0.85	0.77	0.85	0.89	0.88	0.88	0.89	0.90
Average	**75.24**	**82.64**	**85.74**	**76.79**	**86.44**	**89.56**	**90.39**	**90.33**	**90.88**	**92.5**	**0.75**	**0.82**	**0.82**	**0.78**	**0.83**	**0.86**	**0.87**	**0.89**	**0.90**	**0.91**

Table 5: Performance metrics for Class Dependency Network-based SDP models

Projects	Accuracy										AUC									
	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	LightGBM	GAN	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	0.70	0.80	0.81	0.72	0.76	0.79	0.78	0.80	0.80	0.84
Camel	70.01	87.91	75.7	69.58	84.68	82.18	70.13	70.87	71.88	73.69	0.74	0.82	0.81	0.70	0.72	0.77	0.66	0.69	0.70	0.72
Ivy	76.56	86.99	82.57	74.45	82.49	89.76	89.08	89.34	88.71	90.55	0.74	0.80	0.84	0.75	0.80	0.80	0.78	0.87	0.85	0.89
jEdit	74.33	85.11	80.72	76.62	81.96	87.73	86.12	87.41	86.88	87.12	0.73	0.80	0.78	0.73	0.77	0.79	0.76	0.85	0.84	0.84
Lucene	74.85	82.02	83.44	74.19	85.01	88.98	87.91	88.32	89.11	89.99	0.74	0.79	0.76	0.71	0.80	0.78	0.76	0.86	0.86	0.86
Synapse	75.02	79.67	81.67	74.11	83.07	87.31	85.44	84.21	87.92	88.31	0.72	0.76	0.79	0.72	0.82	0.84	0.84	0.83	0.85	0.86
Tomcat	75.82	78.81	80.24	72.68	80.56	85.02	84.67	85.27	86.44	86.91	0.73	0.75	0.78	0.70	0.78	0.83	0.83	0.84	0.84	0.85
Velocity	73.45	79.01	81.45	73.31	83.26	81.78	82.01	82.78	83.55	83.12	0.72	0.74	0.79	0.71	0.80	0.80	0.80	0.80	0.81	0.81
Poi	73.89	82.33	80.67	74.67	84.05	82.37	80.47	81.09	82.47	83.88	0.72	0.80	0.78	0.72	0.83	0.81	0.79	0.79	0.80	0.81
Xalan	76.02	83.71	85.03	74.69	85.44	87.55	85.67	86.21	87.45	88.09	0.71	0.80	0.83	0.72	0.83	0.85	0.84	0.84	0.85	0.85
Average	**74.17**	**82.93**	**81.55**	**74.04**	**83.15**	**85.59**	**83.67**	**84.24**	**85.09**	**86.13**	**0.73**	**0.79**	**0.8**	**0.72**	**0.79**	**0.81**	**0.78**	**0.82**	**0.82**	**0.83**

Table 6: Friedman Test results

Call Graph-based SDP models										
	MNB	LR	RF	KNN	GB	BAGD	MLP	LSTM	LightGBM	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

Table 7 : Performance metrics for software metrics-based SDP models

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**

The performance of various classifiers across the Call Graph-based and Class Dependency Network-based SDP models is statistically compared by the Friedman test results, which are displayed in Table 6. According to the test results, the Generative Adversarial Network classifier obtains the lowest rank for both AUC and accuracy and also for the two graphical representations call graph and class dependency network, indicating that it consistently performs better than other models in all categories, obtaining the lowest mean rank for both the evaluation parameters AUC and accuracy levels. According to this, the Generative Adversarial Network is the best classifier for identifying software defects in all of these graphical representations. The performance of the top-performing classifier is followed by LightGBM, which obtains the second lowest mean rank in the Friedman test results, demonstrating that it is also an effective model for defect prediction. Defects can be accurately predicted by both classifiers, while the Generative Adversarial Network classifier performs somewhat better overall owing to its ability to handle imbalanced defect datasets by generating realistic random samples of synthetic data to handle the class imbalance. The SDP model's performance is contrasted with models that employ software metrics in Figure 2. Based on network metrics obtained from the call graph, the analysis demonstrates an overall improvement in performance for models.

Furthermore, the findings are examined and the overall cost for every dataset is computed. Figure 3 shows the cost analysis (medium testing efficiency) for models created with Software Metrics (SOFM), Call Graph (CG), and Class Dependency Network (CDN). Across all datasets, a similar pattern shows that NEcost values increase with the percentage of defective classes (POFC). Reducing the cost of defect removal compared to conventional testing techniques is essential for accurate and cost-effective defect prediction. Thus, any method of defect removal works best for projects in which the proportion of faulty classes stays below the NCOSTM=1.0 level.

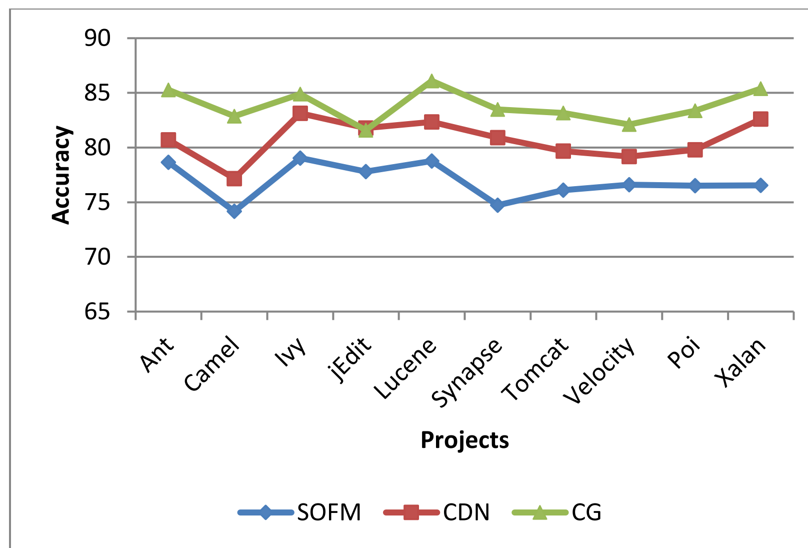

Fig. 2: Comparison of mean prediction accuracy (SOFM: Software Metrics, CDN: Class Dependency Network, CG: Call Graph)

Utilising network metrics obtained from call graphs to forecast software flaws and evaluate related expenses according to testing effectiveness is the main contribution of this study. The majority of approaches have historically concentrated on class dependency networks, which highlight high-level relationships across classes but frequently ignore the specific dependencies within the system. SDP models' performance was assessed in a variety of situations, including within-project, followed by cross-version, and cross-project scenarios, in earlier studies, such as the research conducted by Gong et al. [8]. These research looked at the costs of employing these SDP models and evaluated their efficacy using network measures from class dependency networks, but they did not take testing efficiency across phases into consideration. Although Biçer et al. [20] recommended using network metrics derived from developer connection networks, these studies did not fully analyse the effects of applying various classifiers to classify models as either defect-prone or non-defect-prone. A comparative

analysis of these studies across a number of parameters is provided in Table 8. The cross-validation accuracy and AUC findings, as well as the comparative analysis, highlight how well network metrics derived from call graphs function for defect prediction.

Table 8: Comparative analysis with previous research works

Parameters	Gong [8]	Biçer [20]	Present Work
Network used	Class Dependency Network	Developer Communication Network	Call Graph, Class Dependency Network
Dataset	Groovy, HBase, ActiveMQ, Camel, Hive, JRuby, Derby, Wicket and Lucene	IBM Rational Team Concert, Drupal	Ant, Camel, Ivy, jEdit, Lucene
ML Algorithms	RF, Naive Bayes	Naive Bayes	MNB, LR, RF, KNN, GB, BAGD, MLP, LightGBM, LSTM, GAN
Model Evaluation	Friedman Test, Wilcoxon-signed rank Test, Nemenyi Test	Probability of false alarms and detection	Prediction accuracy and AUC, Friedman Test
Cost Evaluaton	Cost-effectiveness curve and Effort reduction measures	Cost-effectiveness curve	Cost evaluation model comprising testing efficiency in Unit, Integration, System, and Field testing.
SDP context	Within-project, Cross-version, Cross-project	Within-project	Within-project

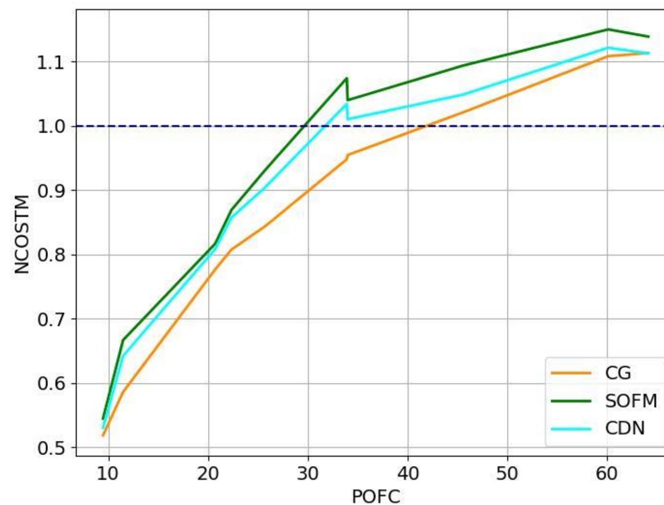

Fig. 3: Cost analysis of the developed models

5. Threats to Validity

This section covers the three main categories of internal, external, and construct validity, which are potential challenges to the validity of our study. Risks for internal validity include the possibility of errors in the defect datasets obtained from the PROMISE repository, variations in survey-derived cost parameters, and the effect of dataset imbalance on prediction results. Using methods like undersampling and oversampling as well as adjusting parameter settings may be necessary to resolve these problems. Though our work focusses on Java projects, further research should confirm that our SDP model is applicable to other programming paradigms and cross-project situations in terms of external validity. The model's emphasis on defect existence without addressing defect quantity or localisation, as well as the impact of network embedding dimensions on performance, are the final threats to construct validity. The efficacy of the model might be improved by investigating different embedding techniques and dimensions.

6. Conclusion and Future Work

This study offers a defect prediction technique that uses software networks—more especially, call graphs—and takes cost into account when predicting software system defects. The proposed model performs better than conventional SDP models that just use software metrics, as well as models that use network metrics from class dependency networks. Through the training of ten classifiers and the generation of network metrics from call graphs, we thoroughly analyse SDP performance and compare it with models based on class dependency networks. The study also assesses how much cost effective the SDP model is in practical usage. This study highlights that the call graph is an excellent graphical representation of software, we evaluate the performance of the suggested model on ten different Java projects. The proposed SDP models improves the AUC ranging from 2.9% to 8.94% for majority of the datasets owing to their ability to better capture the intricate software component relationships which is a critical aspect in SDP. Generative Adversarial Network proved to be the most successful classifier, with the AUC of 0.91 and an accuracy of 92.5% due to its ability to handle imbalanced dataset. The ability of the call graph to depict intricate relationships and dependencies is responsible for the improved defect prediction, which leads to more precise predictions and lowers the cost of software development. The cost-effectiveness of using the proposed SDP framework, is well established through this study as the Figure 3 indicates lowest cost for call graph representation compared to class dependency network and software metrics.

Future work could build on this work by adding data balancing and feature selection techniques, which could improve the SDP model's efficiency by eliminating redundant or irrelevant information from the dataset, resulting in more accurate predictions; furthermore, extending the model's evaluation to include a greater variety of defect datasets across different projects could provide a more thorough understanding of the model's effectiveness and generalisability, which would help validate the model's performance in diverse software environments and ensure its robustness across various contexts.

References

[1] Li, Z., Niu, J., & Jing, X. Y. (2024). Software defect prediction: future directions and challenges. *Automated Software Engineering*, *31*(1), 19, https://doi.org/10.1007/s10515-024-00424-1

[2] Wu, W., Wang, S., Liu, B., Shao, Y., & Xie, W. (2024). A novel software defect prediction approach via weighted classification based on association rule mining. *Engineering Applications of Artificial Intelligence*, *129*, 107622, https://doi.org/10.1016/j.engappai.2023.107622

[3] Boucher, A., & Badri, M. (2018). Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. *Information and Software Technology*, *96*, 38-67, https://doi.org/10.1016/j.infsof.2017.11.005

[4] Khleel, N. A. A., & Nehéz, K. (2024). Software defect prediction using a bidirectional LSTM network combined with oversampling techniques. *Cluster Computing*, *27*(3), 3615-3638, https://doi.org/10.1007/s10586-023-04170-z

[5] Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., & Oliveto, R. (2017). Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, *45*(2), 194-218, https://doi.org/10.1109/TSE.2017.2770122

[6] Chi, J., Qu, Y., Zheng, Q., Yang, Z., Jin, W., Cui, D., & Liu, T. (2018, July). Test case prioritization based on method call sequences. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* (Vol. 1, pp. 251-256). IEEE.

[7] Setia, S., Ravulakollu, K. K., Verma, K., Garg, S., Mishra, S. K., & Sharan, B. (2024, February). Software Defect Prediction using Machine Learning. In *2024 11th International Conference on Computing for Sustainable Global Development (INDIACom)* (pp. 560-566). IEEE.

[8] Gong, L., Rajbahadur, G. K., Hassan, A. E., & Jiang, S. (2021). Revisiting the impact of dependency network metrics on software defect prediction. *IEEE Transactions on Software Engineering*, *48*(12), 5030-5049, https://doi.org/10.1109/TSE.2021.3131950

[9] Kumar, L., Misra, S., & Rath, S. K. (2017). An empirical analysis of the effectiveness of software metrics and fault prediction model for identifying faulty classes. *Computer standards & interfaces*, *53*, 1-32, https://doi.org/10.1016/j.csi.2017.02.003

[10] Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc..

[11] Jureczko, M., & Madeyski, L. (2010, September). Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th international conference on predictive models in software engineering* (pp. 1-10).

[12] Kumar, P. S., & Venkatesan, R. (2020). Improving Software Defect Prediction using Generative Adversarial Networks. *Int. J. Sci. Eng. Appl*, *9*, 117-120.

[13] Arar, Ö. F., & Ayan, K. (2017). A feature dependent Naive Bayes approach and its application to the software defect prediction problem. *Applied Soft Computing*, *59*, 197-209, https://doi.org/10.1016/j.asoc.2017.05.043

[14] Alqarni, A., & Aljamaan, H. (2023). Leveraging Ensemble Learning with Generative Adversarial Networks for Imbalanced Software Defects Prediction. *Applied Sciences*, *13*(24), 13319.

[15] Rebro, D. A., Chren, S., & Rossi, B. (2023, March). Source Code Metrics for Software Defects Prediction. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing* (pp. 1469-1472).

[16] Gupta, S., & Gupta, A. (2017). A set of measures designed to identify overlapped instances in software defect prediction. *Computing*, *99*, 889-914, https://doi.org/10.1007/s00607-016-0538-1

[17] Raamesh, L., Jothi, S., & Radhika, S. (2023). Enhancing software reliability and fault detection using hybrid brainstorm optimization-based LSTM model. *IETE Journal of Research*, *69*(12), 8789-8803, https://doi.org/10.1080/03772063.2022.2069603

[18] Singh, Y., Kaur, A., & Malhotra, R. (2010). Prediction of fault-prone software modules using statistical and machine learning methods. *International Journal of Computer Applications*, *1*(22), 8-15.

[19] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*(6), 476-493.

[20] Biçer, S., Bener, A. B., & Çağlayan, B. (2011, May). Defect prediction using social network analysis on issue repositories. In *Proceedings of the 2011 International Conference on Software and Systems Process* (pp. 63-71).

[21] Wolf, T., Schroter, A., Damian, D., & Nguyen, T. (2009, May). Predicting build failures using social network analysis on developer communication. In *2009 IEEE 31st International Conference on Software Engineering* (pp. 1-11). IEEE.

[22] Nguyen, T. H., Adams, B., & Hassan, A. E. (2010, September). Studying the impact of dependency network measures on software quality. In *2010 IEEE International Conference on Software Maintenance* (pp. 1-10). IEEE.

[23] Zimmermann, T., & Nagappan, N. (2008, May). Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering* (pp. 531-540).

[24] Ulan, M., Löwe, W., Ericsson, M., & Wingkvist, A. (2021). Weighted software metrics aggregation and its application to defect prediction. *Empirical Software Engineering*, *26*(5), 86, https://doi.org/10.1007/s10664-021-09984-2

[25] Ma, W., Chen, L., Yang, Y., Zhou, Y., & Xu, B. (2016). Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology*, *69*, 50-70, https://doi.org/10.1016/j.infsof.2015.09.001

[26] Ponnala, R., & Reddy, C. R. K. (2023). Ensemble model for software defect prediction using method level features of spring framework open source Java Project for E-Commerce. *Journal of Data Acquisition and Processing*, *38*(1), 1645.

[27] Sawadpong, P., & Allen, E. B. (2016, January). Software defect prediction using exception handling call graphs: A case study. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)* (pp. 55-62). IEEE.

[28] Xu, J., Ai, J., & Shi, T. (2021). Software Defect Prediction for Specific Defect Types based on Augmented Code Graph Representation. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)* (pp. 669-678). IEEE.

[29] Gong, L., Rajbahadur, G. K., Hassan, A. E., & Jiang, S. (2021). Revisiting the impact of dependency network metrics on software defect prediction. *IEEE Transactions on Software Engineering*, *48*(12), 5030-5049, https://doi.org/10.1109/TSE.2021.3131950

[30] Qu, Y., Zheng, Q., Chi, J., Jin, Y., He, A., Cui, D., Zhang, H. & Liu, T. (2019). Using k-core decomposition on class dependency networks to improve bug prediction model's practical performance. *IEEE Transactions on Software Engineering*, *47*(2), 348-366, https://doi.org/10.1109/TSE.2019.2892959

[31] Zhou, C., He, P., Zeng, C., & Ma, J. (2022). Software defect prediction with semantic and structural information of codes based on graph neural networks. *Information and Software Technology*, *152*, 107057, https://doi.org/10.1016/j.infsof.2022.107057

[32] Antal, G., Tóth, Z., Hegedűs, P., & Ferenc, R. (2020). Enhanced bug prediction in JavaScript programs with hybrid call-graph based invocation metrics. *Technologies*, *9*(1), 3, https://doi.org/10.3390/technologies9010003

[33] Sharma, D., & Chandra, P. (2024). An empirical analysis of software fault proneness using factor analysis with regression. *Multimedia Tools and Applications*, *83*(17), 52535-52591.

[34] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4), 308-320.

[35] Deng, J., Lu, L., & Qiu, S. (2020). Software defect prediction via LSTM. *IET software*, *14*(4), 443-450, https://doi.org/10.1049/iet-sen.2019.0149