A Systematic Review of Learning-Based Software Defect Prediction

Changjian Li[1], Dongcheng Li[2], Hui Li[1], W. Eric Wong[3], Man Zhao[1]*

[1] School of Computer Science, China University of Geosciences (Wuhan), China
[2] Department of Computer Science, California State Polytechnic University - Humboldt, USA
[3] Department of Computer Science, University of Texas at Dallas, USA
cjlee@cug.edu.cn, dl313@humboldt.edu, huili@vip.sina.com, ewong@utdallas.edu, zhaoman@cug.edu.cn

Abstract

This survey paper on Learning-Based Software Defect Prediction reviews recent advancements in applying machine learning (ML) and deep learning (DL) techniques for software defect prediction (SDP). It covers topics including application scenarios, types of ML/DL, datasets, source code representation, prediction granularity, evaluation metrics, validation methods, and challenges with existing solutions. The paper highlights the increased interest in SDP due to the growing complexity of software and presents a detailed analysis of various ML and DL approaches, their capabilities, and the challenges they present in the context of SDP. It also discusses the evolving nature of these technologies in SDP and their impact on developing reliable and maintainable software systems.

Keywords: Software defect prediction, Deep learning, Machine learning, Predictive model, Quality assurance

1 Introduction

Studies have shown that software defects can cause dramatic failures with severe consequences [1-2]. Thus, software defect prediction has become the cornerstone of quality assurance in the Software Development Life Cycle (SDLC) [3]. With the growing complexity and size of software systems, conventional quality assurance and defect detection techniques are proving to be insufficient. Early identification of software defects not only makes the development process more efficient but also significantly reduces the cost associated with post-deployment fixes [4]. The need for this has greatly increased interest in employing machine learning (ML) and deep learning (DL) technologies for software defect prediction (SDP) [5].

Integrating ML and DL into SDP marks a shift from manual inspection to automated, intelligent prediction models. These models undergo training using historical data on defects and the static attributes of code from prior versions of software, aiming to predict modules with potential defect tendencies. Early application of these models in the SDLC helps prioritize resources and focus on risky modules.

Over the past two decades, there has been a transition from traditional ML to more complex DL methods in SDP. DL, a subset of ML, automatically identifies necessary features from raw data, which is beneficial for detecting complex defect patterns that traditional methods miss. In SDP, this is especially beneficial because the intricate patterns and relationships that could result in defects are often too complicated for traditional algorithms to discern.

Despite DL's advancements in SDP, challenges remain, including data insufficiency, model interpretability, computational demands, and the need for extensive hyperparameter tuning [1]. Moreover, the computational requirements for training deep architectures and the extensive adjustment of hyperparameters present major hurdles.

This review synthesizes current knowledge on ML and DL in SDP, highlighting their capabilities, challenges, and future directions. It aims to provide insights into their impact and how they can enhance software reliability and maintainability.

The following sections of this paper are structured in the following manner: Section 2 presents the preliminaries of the study. Section 3 summarizes related reviews. Section 4 details this study's focus on learning-based SDP, covering datasets, code representations, prediction levels, methods, metrics, validation approaches, challenges, and solutions. The final section outlines conclusions and suggests directions for future research.

2 Background

2.1 Software Defect Prediction

Software Defect Prediction represents a critical field within software engineering, focused on the proactive detection of potential defects in software systems. This predictive approach plays a vital role in enhancing software quality by pinpointing areas where errors might occur, which could evolve into more serious issues during testing or later stages of deployment.

At its core, SDP is about predicting software segments likely to harbor faults, thus enabling early remediation. The nature of this task is inherently complex, requiring high precision in predictions to be genuinely effective. Throughout the years, numerous methods and techniques have been investigated to improve the precision and efficiency of SDP models. Common strategies in SDP include the application of ML algorithms, which have

*Corresponding Author: Dongcheng Li; Email: dl313@humboldt.edu

DOI: https://doi.org/10.70003/160792642025072604009

evolved to include recent DL technologies. These advanced computational methods demonstrate the potential to develop robust SDP models by learning from historical defect data and identifying complex patterns that might not be evidently discernable through traditional analysis.

The general workflow for constructing an SDP model can be summarized into several key steps, as shown in Figure 1:

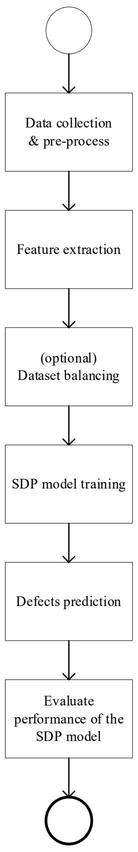

Data collection & pre-process
Feature extraction
(optional) Dataset balancing
SDP model training
Defects prediction
Evaluate performance of the SDP model

Figure 1. General steps for building a SDP model

Beyond the diverse choices in the aforementioned steps, SDP research can be classified by its application scenarios. Traditionally, literature identifies two primary SDP scenarios: Within-Project Defect Prediction (WPDP) and Cross-Project Defect Prediction (CPDP). In WPDP, a project's historical data (such as different versions) is used to predict defective parts [6], meaning WPDP focuses on predicting faults in the same software project where training occurs [7]. Therefore, both the training and test sets originate from the same project. In contrast, CPDP utilizes data from various projects (source projects) to train the SDP model, which is subsequently employed to predict defects in a different project (target project) [8]. As a derivative of transfer learning, this method is particularly useful when there is no labeled data to train on in the target project. In this approach, the primary challenge is reducing the disparity between the source and target projects in terms of feature distribution.

One of the principal challenges in CPDP is the requirement for all projects within the CPDP framework to adhere to identical metric standards. As a counterbalance, heterogeneous defect prediction (HDP) helps predict defects in projects with different metrics by mapping source and target projects' data into a unified frame of reference [9-10].

There are also other widely used approaches beyond these SDP contexts, including JIT-SDP [11], focusing on the prediction of defects at the level of software changes [12]. By integrating JIT-SDP with Change-Level Defect Prediction, developers can identify and fix defects faster, preserving software quality. It is particularly significant in SDP because it offers timely assistance to developers at a more detailed level (change level). It eliminates the need for lengthy code reviews and extensive testing with JIT-SDP [11, 13]. Finally, an emerging SDP methodology is CVDP, which analyzes failure data from previous versions of the same project in order to forecast defects in the current version [14].

2.2 ML in SDP

ML is crucial for Software Defect Prediction, automating defect identification and prediction. In SDP, ML is categorized into supervised and unsupervised learning [2]. By analyzing input-output relationships, supervised learning predicts defects. Unsupervised learning identifies potential defects by analyzing data patterns.

Supervised learning, a common approach in SDP, uses labeled data to train algorithms for reliable category prediction [15]. It predicts unseen instances based on attributes, distinguishing between classification and regression problems based on the label type. In SDP, software instances are data instances with features as attributes and defects as output categories to predict defective modules. Addressing category imbalance with data sampling improves prediction accuracy. Early defect detection using ML enhances software reliability and quality.

2.3 DL in SDP

Deep learning uses layered models to extract complex data representations, which apply to tasks like regression, clustering, classification, ranking, and synthesis. The steps for software defect prediction using deep learning are illustrated in Figure 2.

Deep Neural Networks (DNNs) have three types of layers: input, hidden, and output. The input layer introduces data, hidden layers process it, and the output layer produces results like classification or regression. Layers are usually fully connected, linking every neuron to the next layer's neurons.

Within the scope of defect prediction, researchers are now utilizing various DNN architectures and learning techniques, including Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Deep Autoencoders (DAE), Siamese Neural Networks (SNN), and Long Short-Term Memory (LSTM) networks. These methods are increasingly being used for predicting software defects.

3 Related Works

In recent research in the field of Software Engineering (SE), the application of Machine Learning models has become particularly important. Derived from diverse data formats in SE, these models utilize resources like source code, requirement specifications, and test cases. Specifically, Deep Learning techniques have garnered extensive attention for their applications in Software Defect Prediction. Against this backdrop, this paper seeks to investigate the application and efficacy of ML and DL in addressing SE issues, particularly SDP.

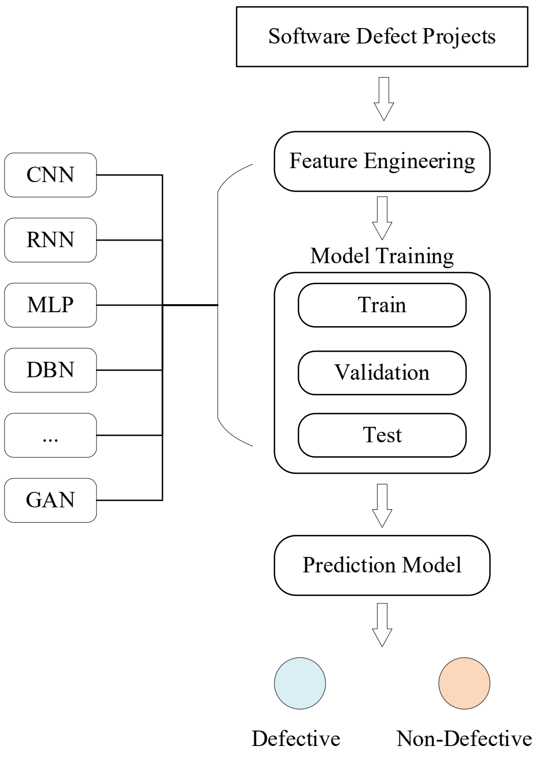


Figure 2. DL-based SDP steps

The SDP and ML methods were comprehensively reviewed by Catal and Diri [16] in their analysis of 74 studies published between 1990 and 2007. Compared to previous years, the number of principal studies increased significantly in 2007. According to Hall et al. [17], 36 studies released between Jan. 2000 and Dec. 2010 were analyzed quantitatively and qualitatively.

Malhotra [18] reviewed 64 studies spanning from 1991 to 2013 to explore the application of ML in SDP. Son et al. [19] carried out a systematic mapping study of 156 research papers, examining the use of ML and DL on SDP. DL's use in predicting software quality was reviewed by Malhotra et al. [20]. Pandey et al. [21] published their results in their systematic review of the application of ML and DL on SDP. Giray et al. [22] published a review focused on the application of DL in SDP. Sharma et al. [23] examined the use of integrated machine learning in software defect prediction from 2018 to 2021. Nevendra and Singh [24] delved into the deep learning techniques used over the past six years and conducted comparative research on software defect prediction at both file and change levels. Prabha and Shivakumar [25] investigated various ML-based SDP techniques in industrial applications and proposed corresponding hybrid approaches. Matloob [26] specifically studied supervised machine learning techniques aimed at improving performance and reducing costs.

This study differs from related research in the following ways: (1) it comprehensively focuses on the latest applications of DL and ML in SDP in recent years; (2) it delves into multiple aspects; (3) it covers a wide range of literature, including 112 main studies published up to mid-2023; (4) it employs a systematic review on research methods.

4 Discussion

4.1 Paper Collection Methodology and Result

We performed precise keyword searches in databases like Google Scholar, DBLP, arXiv, and IEEE Xplore to gather high-quality sources on learning-based software defect prediction. After screening over 50 searches and 300 papers, we selected 112 papers that reflect the high quality and current state of this technology.

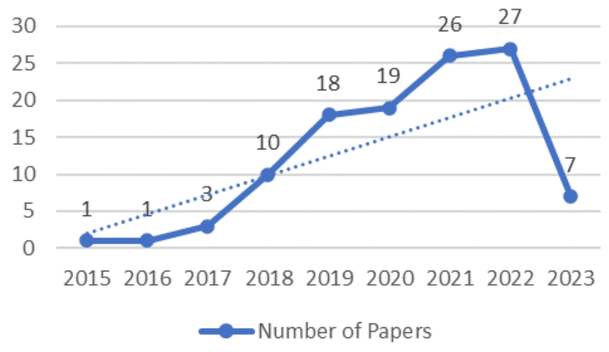


Figure 3. Learning-based SDP Publications during 2015-2023

Figure 3 shows the publication trend of the 112 high-quality papers we collected, with solid dots indicating annual publications and dashed lines showing trends. Few papers were published before 2018 due to the limited use of ML and DL. From 2018 to 2022, research in this area surged. The data from 2023 is insufficient for trend analysis.

4.2 Training and Testing Dataset

The datasets utilized in each study were extracted and are enumerated in Table 1, with most being developed in programming languages like Java, C, and C++. The most frequently used dataset is from the PROMISE repository [27], which is a dataset developed in Java language. Some studies, such as Huang et al. [28] used datasets for Android mobile applications developed in Java, Kotlin, JavaScript, and C++, like Android Firewall, Android SDK, etc. Dong et al. [29] concentrated on forecasting errors in Android binary executables, commonly referred to as 'apk'.

They sourced Android projects from GitHub, including Wikipedia and chess applications, and built datasets for defect prediction. The experimental methods developed by Xu et al. [30] and Zhao et al. [31-32] have also been applied to Android projects in developing and evaluating error prediction models.

Table 1. Data sets applied in 10 more studies

Project	Nums	Programming language
camel	57	Java
xalan	54	Java
xerces	50	Java
poi	49	Java
log4j	46	Java
lucene	45	Java
synapse	45	Java
jedit	44	Java
ant	40	Java
ivy	33	Java
velocity	22	Java
pc1	22	C
cm1	20	C
kc1	20	C++
eclipse jdt	19	Java
pc3	18	C
pc4	18	C
jm1	18	C
mw1	17	C
kc2	15	C++
mc1	14	C++
pc2	12	C
mc2	12	C

4.3 Representation of Source Code

In contrast to source code, DL algorithms operate on numerical vectors. In order to represent source code, we need to transform it into a format that can be analyzed by DL algorithms. This conversion process must aim to minimize information loss during the transformation. Figure 7 shows the methods of source code representation adopted in our pool of major studies. The representation techniques in SDP are heavily influenced by a variety of metrics, including software structure and size, as well as product and process metrics. 40 studies utilized intermediate representations (some involving multiple types), such as AST, assembly code, CFGs, PDGs, DFGs, and images, to form numerical vectors processable by DL algorithms. In 8 studies, metrics were integrated with intermediate representations. Two studies directly converted source code into numerical vectors.

67 studies employed a set of metrics for representing source code. Xu et al. [33] utilized the CKJM tool [34] to extract software sizes and fundamental metrics from the bytecode for generated Java documents in their investigation. These metrics include the number of methods per class, the level of connectivity among object classes, as well as McCabe's cyclomatic complexities. Characteristics of processes or changes, derived from the change history of software projects, act as indicators in defect prediction [35]. For example, Yang et al. [36] used change metrics like the number of modified files, the count of developers modifying these files, and the lines of code added or removed, for JIT-DP. Ardimento et al. [37] employed many measures, such as submit rate, developer experience, authored commits, and the average interval among contributions, to describe the method of development. Another metric type used in defect prediction is product metrics, which describe the quality of the source code's internal structure. These metrics include the number of inherited properties, the width of the inheritance tree, the count of methods, and the quantity of methods that are static. Some researchers, including Tong et al. [38] and Zhao et al. [39], standardized metric values prior to creating numerical vectors.

The Abstract Syntax Tree (AST) represents the abstract syntax framework of source-code as a tree (Mou et al. [40]). Thirty-eight studies employed AST as an intermediate representation to develop numerical vectors. Liang et al. [41] converted source code into AST and generated token sequences by extracting tokens from AST nodes. Chen et al. [8] utilized a streamlined version of Abstract Syntax Tree (AST), where they disregarded node types that were not relevant to the project and excluded project-specific function and parameter names. Researchers like Li et al. [42], Dam et al. [43], and Liu et al. [44] utilized word embeddings to derive numerical vectors from ASTs. Shi et al. [45] used a source code representation method based on AST path pairs, PathPair2Vec, to build embedding vectors.

Li et al. [46] modeled and analyzed the relationships between AST paths of different methods using PDG and DFG. While buggy paths in the AST represent the local context of buggy code, the global context is depicted through the relationships between buggy methods, illustrated by program and data flow dependencies. Phan and Nguyen [47] favored using assembly instruction sequences rather than ASTs, arguing that they more effectively simulate program behavior because they are closer to machine code and better reflect program structure. Phan et al. [48] built control flow graphs from assembly instructions derived from compiled source code. According to Chen et al. [49], source code can be visualized as images and defects are predicted using an image classification model. An image was then generated from this vector for classification by a pre-trained ImageNet AlexNet model. Eight studies integrated inputs based on ASTs with a collection of metrics. Fan et al. [50], Li et al. [42], Lin and Lu [51], Qiu et al. [52], Shi et al. [53], and Wang et al. [54] combined word embeddings from ASTs with metrics.

In two studies, source code was directly transformed into numerical vectors without using intermediate representations. Hoang [55] et al. employed NLTK to parse commit messages and code changes, representing each word in the commit information and code changes as n-dimensional vectors. Tian and Tian [56] used Word2vec (Mikolov et al. [57]) to convert source code into fixed-length vectors.

4.4 Granularity Level of Prediction

Predicting software defects at various granularity levels is the goal of constructing machine learning/ deep learning models (Nam et al. [58]), and these levels include file, module, change, category, function, program, and statement. Previous studies have shown that the granularity level of prediction not only affects the predictive performance of the model but also impacts the effort required to locate defects (Koru and Liu [59]; Calikli et al. [60]). There were 41 studies dedicated to file-level defect prediction models, and 33 studies focused on the module level. 13 studies were concerned with the change level. Four studies dealt with category-level predictions in software systems created with object-oriented programming languages. Three studies each targeted finer-grained predictions at the method or line level (Refer to Figure 4).

DL has become increasingly popular for finer-grained SDP over the last few years, in keeping with Kamei and Shihab's observations [61]. Since 2019, ten studies (two in 2019, four in 2020, three in 2021, one in 2022) have reported their experimental results in category, statement, and program-level predictions. The application of DL algorithms to change-level defect prediction has also been published by researchers since 2018. Over the last four years, there have been 12 follow-up studies: three in 2020, four in both 2019 and 2021, and one in 2022.

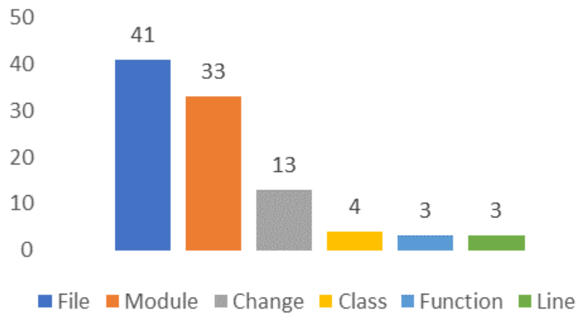


Figure 4. Distribution of the granularity level of prediction

4.5 DL Approaches in SDP

As shown in Figure 5, the most used deep learning model in the SDP field is the Convolutional Neural Network (CNN) model. Besides CNN, other widely used deep learning models in SDP are LSTM/RNN, MLP, and Deep Belief Networks (DBN).

As a classic model in the deep learning field, Convolutional Neural Networks perform exceptionally well in SDP, capturing local patterns in high-dimensional data (Li et al. [42]; Pan et al. [62]) and demonstrating good scalability when combined with structures like Transformers (Liu et al. [63]) or GRU (Khleel et al. [64]). RNN and LSTM architectures are also chosen by many researchers due to their ability to capture long-range dependencies (Uddin et al. [65]). DBN is used to learn representations that can reconstruct training data with high probability.

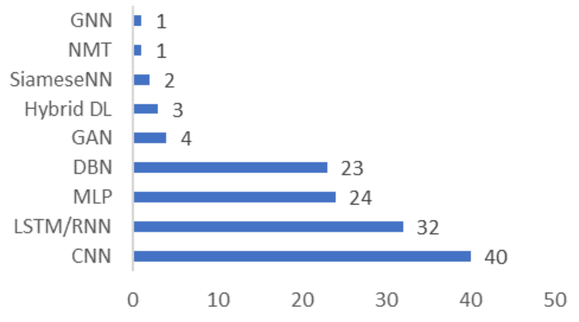


Figure 5. Distribution of the DL approaches

Other models such as GAN and GNN are less frequently used in the SDP field. Researchers choose these models for specific purposes, such as using GAN to create synthetic data (Sun et al. [66]), using Siamese networks for limited data (Zhao et al. [67]), and capturing source code ASTs in GNN (Xu et al. [68]).

4.6 ML Approaches in SDP

As shown in Figure 6, the most used machine learning model in the SDP field is the SVM, followed by ANN, DT, and NB models. SVM is a widely used classification technique in machine learning, adept at handling high-dimensional data and complex nonlinear relationships, and performs particularly well on small sample datasets. (Liu et al. [69]) used an improved dual support vector machine technique, yielding excellent results compared to other linear and nonlinear techniques. ANN, which simulates the neuronal connections of the human brain with multiple layers of neurons, can construct complex nonlinear models, hence its widespread adoption by researchers (Albahli et al. [70]; Goyal et al. [71]). Decision tree models are generally used in classification scenarios where data features have a clear physical meaning, while Naive Bayes is often chosen by researchers for its simplicity. Notably, the integration of multiple machine learning algorithms has become increasingly common in recent technologies, such as Khalid et al. [72] combining particle swarm optimization for model enhancement, and Goyal et al. [73] proposing a feature selection method using genetic evolutionary technology to enhance the accuracy of SVM-based software defect prediction classifiers by identifying the optimal feature subsets.

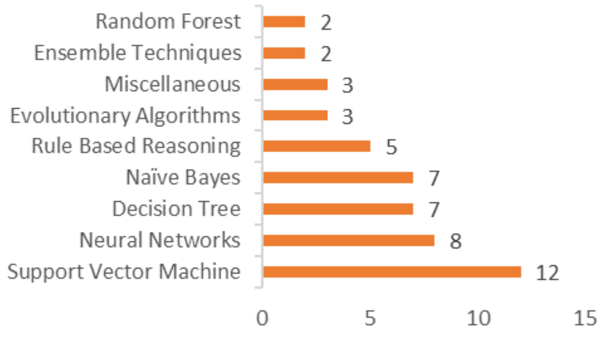


Figure 6. Distribution of the ML approaches

Methods like evolutionary algorithms, rule-based reasoning, and ensemble techniques are not as frequently used in SDP, yet notably, there has been a growing trend among researchers to employ random forest algorithms for SDP studies, exemplified by Zheng et al [13].

4.7 Evaluation Metrics and Validation Methods

Researchers typically use various evaluation metrics and validation methods to assess defect prediction models. Figure 7 shows the 11 most commonly used evaluation metrics in the papers we collected.

Accuracy and error rate are key metrics for evaluating classification models. Accuracy ranges from 0 to 1, with 0 and 1 indicating completely erroneous or correct predictions, respectively. The error rate is the inverse of accuracy, indicating the percentage of flaws in predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad \textbf{(1)}$$

$$Error\ Rate = 1 - Accuracy \quad \textbf{(2)}$$

Precision, the percentage of classes correctly identified as defective out of those classified as defective, measures our effectiveness in detecting defects. Recall is the ratio of actual defects in the categories predicted as defective, indicating the extent to which we might overlook defective categories.

$$Precision = \frac{TP}{TP + FP} \quad \textbf{(3)}$$

$$Recall = TPR = Sensitivity = \frac{TP}{TP + FN} \quad \textbf{(4)}$$

The F-measure, which is the harmonic mean of precision and recall, ranges from 0 to 1, with higher values signifying better performance. G-measure combines the true positive and true negative rates. MCC (Matthews Correlation Coefficient) ranges from -1 to 1, with values of 1, 0, and -1 indicating perfect, random, and incorrect predictions, respectively.

$$F - Measure = \frac{\left(1 + \beta^2\right) * Recall * Precision}{\beta^2 * Recall + Precision} \quad \textbf{(5)}$$

$$G - Measure = \frac{2 * TPR * TNR}{TPR + TNR} \quad \textbf{(6)}$$

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad \textbf{(7)}$$

68% of researchers used the F1 score to assess balance between precision and recall; 55 studies used recall, assessing defect prediction; 48 used precision to reflect correctly predicted defects. AUC was used in 44 studies to evaluate classification ability; 38 studies used accuracy, suitable for imbalanced datasets. Additionally, 23 studies used MCC to address class imbalance. Effort-aware metrics in software engineering, used in 13 studies, assess defect prediction considering the cost of identifying and fixing defects, emphasizing both accuracy and practical implications in resource allocation.

Additionally, 11 studies used G-measure to balance detection and false positive rates; 9 used ROC curve for trade-off analysis between true and false positives; 5 used mean squared error to assess average error and accurately predicted non-defects; 2 used AUC-PR curve to evaluate precision-recall trade-offs.

Regarding validation methods, 65 studies used the holdout method to divide the original dataset into training and testing sets, while 47 studies preferred the cross-validation method.

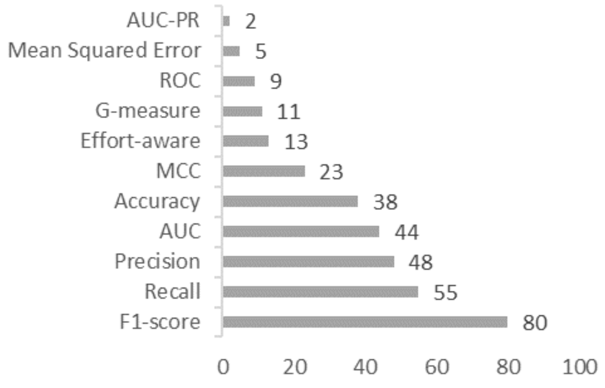


Figure 7. Distribution of the evaluation metrics

4.8 Research Reproducibility

A key characteristic of scientific research is the reproducibility of its results (González-Barahona & Robles [74]). For a study to be considered reproducible, other researchers should be able to replicate the experimental results reported in the study using the tools (including source code and datasets) provided by the original authors in a similar experimental setting (Liu et al. [75]). Several academics have highlighted the reproducibility issue within the software engineering domain (Lewowski & Madeyski [76]). Recently, Liu et al. [77] reviewed studies on the application of deep learning models to software engineering challenges, including defect prediction and code clone detection. They discovered that over half of the studies did not provide high-quality source code or comprehensive datasets, which are essential for reproducing their deep learning models. Consequently, we assessed whether the primary studies had published packages that support reproducibility, using the categorization method employed by Lewowski and Madeyski [76].

Figure 8 shows the existence of reproducibility packages in studies of our researched paper collection. It was observed that 54 articles (accounting for 48% of studies) made no reference to any kind of reproducibility package. Four studies claimed to provide these packages, but the provided links were either missing or inaccessible. Additionally, 35 studies furnished solely data, and two provided exclusively scripts for replication. Merely 17 publications (comprising 15% of studies) contained comprehensive data and scripts. Moreover, we did not observe a marked increase in the trend of sharing reproducibility packages.

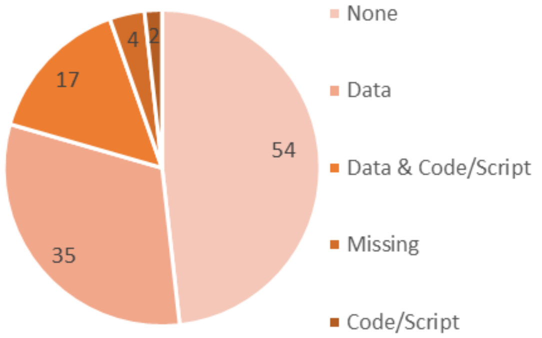


Figure 8. Distribution of the reproducibility packages

4.9 Challenges and Solutions

The goals of this section are to summarize what has been mentioned in the collected papers regarding challenges and solutions. We focus on the specific challenges of using deep learning or machine learning models in the SDP phase of software development, including new issues that have not been extensively explored in this field. We categorize these challenges into three main types: model, data, and universal challenges, and summarize the solutions to these problems found in existing research.

4.9.1 Model

• Overfitting and performance decline simulation: Overfitting and performance decline are potential challenges. Employing appropriate techniques (such as dropout regularization) and updating models based on user feedback can address these issues.

• Semantic features and feature redundancy: Traditional features may be insensitive to the semantics of the program and have feature redundancy. Solutions include using a richer feature set, such as code comments, embeddings, AST, and structural features, as well as feature selection using deep learners and meta-heuristic methods.

• Manual feature selection and context dependency: Manual feature selection might lead to bias, and different feature sets perform differently in various contexts. Self-attention mechanisms and specific LSTM architectures can optimize these issues.

• Random parameter initialization and fixed-length feature vectors: Randomly chosen parameters might reduce performance, while traditional classification algorithms assume all feature vectors are of equal length. Meta-heuristic methods and the use of latent features with variable lengths can solve these problems.

• Sequence network sensitivity to hyperparameters: Sequence networks fail to fully capture the tree structure and semantic dependencies of ASTs, and different hyperparameter settings can lead to performance variations. Non-sequential networks like Bidirectional LSTM and HNN, along with the application of meta-heuristic methods, can tackle these challenges.

4.9.2 Data

• Heterogeneous data and dataset size limitations: The data utilized in SDP is frequently highly heterogeneous and limited in size. Solutions include adopting various deep learning architectures to adapt to data differences, and introducing standardization and transformation steps in data preprocessing and feature extraction. Also, expanding dataset sizes to enhance result accuracy is a future challenge.

• Insufficient training data and data imbalance: Restricted training data and the scarcity of software defects lead to data imbalance, affecting model performance. Potential solutions include employing deep learning architectures that can learn from limited data and implementing data balancing techniques within the models.

• Formation of training data and incomplete code fragments: Training data may contain invalid defect instances, and dealing with incomplete code fragments is challenging. These issues can be addressed by manually verifying training data and using heuristic methods to extract information from incomplete code.

4.9.3 Deep Learning

• Data Dependency: Deep learning algorithms gradually learn through datasets. Employing extensive datasets guarantees that learning algorithms produce expected outcomes. A powerful learning algorithm requires adjustment of many parameters, and fine-tuning key parameters requires a large amount of data. However, artificial neural networks demand substantial data volumes for effective learning. Furthermore, most datasets in this survey are relatively small because manual labeling, external sources, and delegated classification are required.

• Hyperparameter Optimization: Hyperparameters refer to the parameters of a learning algorithm that are set prior to the initiation of the learning process. Slight alterations in the values of these parameters can significantly enhance the performance of the learning model. However, identifying the optimal hyperparameters poses a challenge, as it tends to extend the duration of the training process and necessitates considerable training resources and human expertise.

• Requires High-Performance Configuration: To improve the performance of deep learning algorithms, a large number of samples is necessary. However, efficient handling of real datasets also demands high processing capabilities. It is possible for data scientists to achieve significant performance gains in shorter timeframes by using multi-core, high-performance GPUs and similar processing units. As a result, these units are expensive and consume a lot of electricity.

• Blackbox: Our models and the data that feed neural networks are well understood, but we do not fully understand how this data is processed. Theoretical comprehension of how specific results are derived remains elusive. Neural networks essentially function as black boxes, making it challenging for scientists to discern how conclusions are reached. Table 2 lists software packages that offer deep learning implementations, with different columns representing application names, licenses, supported platforms, and provided interface information.

Table 2. Opensource software that provide DL implementations

Name	License	Platform	Interface
BigDL	Apache2.0	Spark	Scala, Python
Caffe	BSD	Linux, MacOS, Windows	Python, Matlab, C++
DL4j	Apache2.0	Linux, MacOS, Windows, Android	Java, Scala, Clojure, Python, Kotlin
Dlib	Boost Software License	Cross-platform	C++
Intel DAAL	Apache2.0	Linux, MacOS, Windows with Intel CPU	C++, Python, Java
Keras	MIT license	Linux, MacOS, Windows	Python, R
Microsoft CNTK	MIT license	Windows, Linux, MacOS with docker	Python, C++, BrainScript, Shell
Apache MXNet	Apache2.0	Linux, MacOS, Windows, Android, AWS, IOS	C++, Python, Julia, Matlab, JavaScript, Go, R, Scala Prel, Clojure
OpenNN	GNU LGPL	Cross-platform	C++
PlaidML	AGPL	Linux, MacOS, Windows	Python, C++
Pytorch	BSD	Linux, MacOS, Windows	Python, C++
Apache SINGA	Apache2.0	Linux, MacOS, Windows	Python, C++, Java
TensorFlow	Apache2.0	Linux, MacOS, Windows, Android	Python, C/C++, Java, Go, JavaScript, R, Julia, Swift
Theano	BSD	Cross-platform	Python
Torch	BSD	Linux, MacOS, Windows, Android, IOS	Lua, LuaJIT, C/C++

4.9.4 Universal

• Lack of utilizing defect-related information beyond code: Current methods do not fully leverage defect-related information beyond code, such as comments and commit details. Exploring this area presents a future challenge.

• Software quality and security: Issues in software quality and security pose significant threats to organizations. Timely prediction and identification of software defects are crucial for delivering reliable software products.

• Effectiveness of feature representation and class imbalance issues: Effectively representing software defect features is challenging, and class imbalance issues within datasets need to be addressed.

• Experimental dataset selection: Choosing appropriate datasets, considering factors like data imbalance, size, and defect rates, is crucial to ensure the generality of experimental results.

5 Conclusion

Software Defect Prediction utilizes a variety of techniques to automatically identify defects in software, which aids in reducing the effort required to rectify these defects. In today's context of continually increasing software quantities and relatively limited quality assurance resources, this technology is particularly important. Especially in recent years, Software Defect Prediction based on learning algorithms has received considerable attention. Our study involved an extensive literature review of SDP methods employing machine learning and deep learning technologies, aiming to capture the latest advancements in these areas. These studies include both quantitative and qualitative analyses, covering various aspects of SDP: application scenarios, types of machine learning/deep learning, datasets, source code representation, granularity of prediction, evaluation metrics, validation methods, as well as proposed solutions and challenges.

The findings reveal an uptick in SDP research recently, incorporating various techniques, datasets, and validation methods. We have compiled insights on data, models, and the prevalent challenges in SDP, summarizing some solutions proposed by the researchers. These research outcomes are helpful for newcomers to the SDP field, as they help in understanding the scope of different methods. For experienced researchers, they assist in focusing their research directions for the coming years.

References

[1] W. E. Wong, X. Li, P. A. Laplante, Be More Familiar with Our Enemies and Pave the Way Forward: A Review of the Roles Bugs Played in Software Failures, *Journal of Systems and Software*, Vol. 133, pp. 68-94, November, 2017.

[2] W. E. Wong, V. Debroy, A. Surampudi, H. J. Kim, M. F. Siok, Recent Catastrophic Accidents: Investigating How Software Was Responsible, *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement*, Singapore, 2010, pp. 14-22.

[3] D. K. Yadav, S. K. Chaturvedi, R. B. Misra, Early Software Defects Prediction Using Fuzzy Logic, *International Journal of Performability Engineering*, Vol. 8, No. 4, pp. 399-408, July, 2012.

[4] W. E. Wong, J. R. Horgan, M. Syring, W. Zage, D. Zage, Applying Design Metrics to Predict Fault-Proneness: A Case Study on a Large-Scale Software System, *Software: Practice and Experience*, Vol. 30, No. 14, pp. 1587-1608, November, 2000.

[5] R. Bhandari, S. Singla, P. Sharma, S. S. Kang, AINIS: An Intelligent Network Intrusion System, *International Journal of Performability Engineering*, Vol. 20, No. 1, pp. 24-31, January, 2024.

[6] S. Omri, C. Sinz, Deep Learning for Software Defect Prediction: A Survey, *IEEE/ACM 42nd International Conference on Software Engineering Workshops*, Seoul, South Korea, 2020, pp. 209-214.

[7] C. Ni, W. S. Liu, X. Chen, Q. Gu, D. X. Chen, Q. G. Huang, A Cluster Based Feature Selection Method for Cross-Project Software Defect Prediction, *Journal of Computer Science and Technology*, Vol. 32, No. 6, pp. 1090-1107, November, 2017.

[8] D. Chen, X. Chen, H. Li, J. Xie, Y. Mu, DeepCPDP: Deep Learning Based Cross-Project Defect Prediction, *IEEE Access*, Vol. 7, pp. 184832-184848, December, 2019.

[9] H. Chen, X. Y. Jiang, Y. Zhou, B. Li, B. Xu, Aligned Metric Representation Based Balanced Multiset Ensemble Learning for Heterogeneous Defect Prediction, *Information and Software Technology*, Vol. 147, Article No. 106892, July, 2022.

[10] J. Nam, S. Kim, Heterogeneous Defect Prediction, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, 2015, pp. 508-519.

[11] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A Large-Scale Empirical Study of Just-In-Time Quality Assurance, *IEEE Transactions on Software Engineering*, Vol. 39, No. 6, pp. 757-773, June, 2013.

[12] G. G. Cabral, L. L. Minku, E. Shihab, S. Mujahid, Class Imbalance Evolution and Verification Latency in Just-In-Time Software Defect Prediction, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, 2019, pp. 666-676.

[13] W. Zheng, T. Shen, X. Chen, P. Deng, Interpretability Application of the Just-In-Time Software Defect Prediction Model, *Journal of Systems and Software*, Vol. 188, Article No. 111245, June, 2022.

[14] J. Zhang, J. Wu, C. Chen, Z. Zheng, M. R. Lyu, CDS: A Cross-Version Software Defect Prediction Model with Data Selection, *IEEE Access*, Vol. 8, pp. 110059-110072, June, 2020.

[15] X. Chen, D. Zhang, Y. Zhao, Z. Cui, C. Ni, Software Defect Number Prediction: Unsupervised vs. Supervised Methods, *Information and Software Technology*, Vol. 106, pp. 161-181, February, 2019.

[16] C. Catal, B. Diri, A Systematic Review of Software Fault Prediction Studies, *Expert Systems with Applications*, Vol. 36, No. 4, pp. 7346-7354, May, 2009.

[17] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A Systematic Literature Review on Fault Prediction Performance in Software Engineering, *IEEE Transactions on Software Engineering*, Vol. 38, No. 6, pp. 1276-1304, November-December, 2012.

[18] R. Malhotra, A Systematic Review of Machine Learning Techniques for Software Fault Prediction, *Applied Soft Computing*, Vol. 27, pp. 504-518, February, 2015.

[19] L. H. Son, N. Pritam, M. Khari, R. Kumar, P. T. M. Phuong, P. H. Thong, Empirical Study of Software Defect Prediction: A Systematic Mapping, *Symmetry*, Vol. 11, No. 2, Article No. 212, February, 2019.

[20] R. Malhotra, S. Gupta, T. Singh, A Systematic Review on Application of Deep Learning Techniques for Software Quality Predictive Modeling, *2020 International Conference on Computational Performance Evaluation (ComPE)*, Shillong, India, 2020, pp. 332-337.

[21] S. K. Pandey, R. B. Mishra, A. K. Tripathi, Machine Learning Based Methods for Software Fault Prediction: A Survey, *Expert Systems with Applications*, Vol. 172, Article No. 114595, June, 2021.

[22] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, B. Tekinerdogan, On the Use of Deep Learning in Software Defect Prediction, *Journal of Systems and Software*, Vol. 195, Article No. 111537, January, 2023.

[23] T. Sharma, A. Jatain, S. Bhaskar, K. Pabreja, Ensemble Machine Learning Paradigms in Software Defect Prediction, *Procedia Computer Science*, Vol. 218, pp. 199-209, 2023.

[24] M. Nevendra, P. Singh, A Survey of Software Defect Prediction Based on Deep Learning, *Archives of Computational Methods in Engineering*, Vol. 29, No. 7, pp. 5723-5748, November, 2022.

[25] C. L. Prabha, N. Shivakumar, Software Defect Prediction Using Machine Learning Techniques, *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)*, Tirunelveli, India, 2020, pp. 728-733.

[26] F. Matloob, S. Aftab, M. Ahmad, M. A. Khan, A. Fatima, M. Iqbal, W. M. Alruwaili, N. S. Elmitwally, Software defect prediction using supervised machine learning techniques: a systematic literature review, *Intelligent Automation & Soft Computing*, Vol. 29, No.2, pp. 403-421, June, 2021.

[27] M. Jureczko, L. Madeyski, Towards Identifying Software Project Clusters with Regard to Defect Prediction, *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, Timişoara, Romania, 2010, pp. 1-10.

[28] Q. Huang, Z. Li, Q. Gu, Multi-task Deep Neural Networks for Just-in-Time Software Defect Prediction on Mobile Apps, *Concurrency and Computation: Practice and Experience*, Vol. 36, No. 10, Article No. e7664, May, 2024.

[29] F. Dong, J. Wang, Q. Li, G. Xu, S. Zhang, Defect Prediction in Android Binary Executables Using Deep Neural Network, *Wireless Personal Communications*, Vol. 102, No. 3, pp. 2261-2285, October, 2018.

[30] Z. Xu, K. Zhao, T. Zhang, C. Fu, M. Yan, Z. Xie, X. Zhong, G. Catolino, Effort-aware Just-in-Time Bug Prediction for Mobile Apps via Cross-Triplet Deep Feature Embedding, *IEEE Transactions on Reliability*, Vol. 71, No. 1, pp. 204-220, March, 2022.

[31] K. Zhao, Z. Xu, M. Yan, Y. Tang, M. Fan, G. Catolino, Just-in-Time Defect Prediction for Android Apps via Imbalanced Deep Learning Model, *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, Virtual Event, Republic of Korea, 2021, pp. 1447-1454.

[32] K. Zhao, Z. Xu, M. Yan, L. Xue, W. Li, G. Catolino, A Compositional Model for Effort-Aware Just-In-Time Defect Prediction on Android Apps, *IET Software*, Vol. 16, No. 3, pp. 259-278, June, 2022.

[33] Z. Xu, S. Li, J. Xu, J. Liu, X. Luo, Y. Zhang, T. Zhang, J. Keung, Y. Tang, LDFR: Learning Deep Feature Representation for Software Defect Prediction, *Journal*

of Systems and Software, Vol. 158, Article No. 110402, December, 2019.

[34] D. Spinellis, Tool Writing: A Forgotten Art? (Software Tools), *IEEE Software*, Vol. 22, No. 4, pp. 9-11, July-August, 2005.

[35] F. Rahman, P. Devanbu, How, and Why, Process Metrics Are Better, *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, 2013, pp. 432-441.

[36] X. Yang, D. Lo, X. Xia, Y. Zhang, J. Sun, Deep Learning for Just-in-Time Defect Prediction, *2015 IEEE International Conference on Software Quality, Reliability and Security*, Vancouver, BC, Canada, 2015, pp. 17-26.

[37] P. Ardimento, L. Aversano, M. L. Bernardi, M. Cimitile, M. Iammarino, Just-in-Time Software Defect Prediction Using Deep Temporal Convolutional Networks, *Neural Computing and Applications*, Vol. 34, No. 5, pp. 3981-4001, March, 2022.

[38] H. Tong, B. Liu, S. Wang, Software Defect Prediction Using Stacked Denoising Autoencoders and Two-Stage Ensemble Learning, *Information and Software Technology*, Vol. 96, pp. 94-111, April, 2018.

[39] L. Zhao, Z. Shang, L. Zhao, T. Zhang, Y. Y. Tang, Software Defect Prediction via Cost-Sensitive Siamese Parallel Fully-Connected Neural Networks, *Neurocomputing*, Vol. 352, pp. 64-74, August, 2019.

[40] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, Convolutional Neural Networks Over Tree Structures for Programming Language Processing, *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30, No. 1, pp. 1287-1293, February 2016.

[41] H. Liang, Y. Yu, L. Jiang, Z. Xie, Seml: A Semantic LSTM Model for Software Defect Prediction, *IEEE Access*, Vol. 7, pp. 83812-83824, June, 2019.

[42] J. Li, P. He, J. Zhu, M. R. Lyu, Software Defect Prediction via Convolutional Neural Network, *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Prague, Czech Republic, 2017, pp. 318-328.

[43] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, C. J. Kim, Lessons Learned from Using a Deep Tree-Based Model for Software Defect Prediction in Practice, *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, Montreal, QC, Canada, 2019, pp. 46-57.

[44] Q. Liu, J. Xiang, B. Xu, D. Zhao, W. Hu, J. Wang, Aging-Related Bugs Prediction via Convolutional Neural Network, *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*, Xi'an, China, 2020, pp. 90-98.

[45] K. Shi, Y. Lu, J. Chang, Z. Wei, PathPair2Vec: An AST Path Pair-Based Code Representation Method for Defect Prediction, *Journal of Computer Languages*, Vol. 59, Article No. 100979, August, 2020.

[46] H. Li, X. Li, X. Chen, X. Xie, Y. Mu, Z. Feng, Cross-project defect prediction via ASTToken2Vec and BLSTM-based neural network, *2019 International Joint Conference on Neural Networks (IJCNN)*, Budapest, Hungary, 2019, pp. 1-8.

[47] A. V. Phan, M. Le Nguyen, Convolutional neural networks on assembly code for predicting software defects, *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*, Hanoi, Vietnam, 2017, pp. 37-42.

[48] A. V. Phan, M. Le Nguyen, L. T. Bui, Convolutional neural networks over control flow graphs for software defect prediction, *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, Boston, MA, USA, 2017, pp. 45-52.

[49] J. Chen, K. Hu, Y. Yu, Z. Chen, Q. Xuan, Y. Liu, V. Filkov, Software visualization and deep transfer learning for effective software defect prediction, *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Seoul South Korea, 2020, pp. 578-589.

[50] G. Fan, X. Diao, H. Yu, K. Yang, L. Chen, Deep semantic feature learning with embedded static metrics for software defect prediction, *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Putrajaya, Malaysia, 2019, pp. 244-251.

[51] J. Lin, L. Lu, Semantic Feature Learning via Dual Sequences for Defect Prediction, *IEEE Access*, Vol. 9, 13112-13124, January, 2021.

[52] S. Qiu, H. Xu, J. Deng, S. Jiang, L. Lu, Transfer Convolutional Neural Network for Cross-Project Defect Prediction, *Applied Sciences*, Vol. 9, No. 13, Article No. 2660, July, 2019.

[53] K. Shi, Y. Lu, G. Liu, Z. Wei, J. Chang, Mpt-Embedding: An Unsupervised Representation Learning of Code for Software Defect Prediction, *Journal of Software: Evolution and Process*, Vol. 33, No. 4, Article No. e2330, April, 2021.

[54] H. Wang, W. Zhuang, X. Zhang, Software Defect Prediction Based on Gated Hierarchical LSTMs, *IEEE Transactions on Reliability*, Vol. 70, No. 2, pp. 711-727, June, 2021.

[55] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, N. Ubayashi, DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction, *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, Montreal, QC, Canada, 2019, pp. 34-45.

[56] J. Tian, Y. Tian, A Model Based on Program Slice and Deep Learning for Software Defect Prediction, In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, Honolulu, HI, USA, 2020, pp. 1-6.

[57] T. Mikolov, K. Chen, G. Corrado, J. Dean, *Efficient Estimation of Word Representations in Vector Space*, September, 2013. https://arxiv.org/abs/1301.3781

[58] J. Nam, *Survey on Software Defect Prediction*, Technical Report, 2014.

[59] A. G. Koru, H. Liu, An Investigation of the Effect of Module Size on Defect Prediction Using Static Measures, *Proceedings of the 2005 Workshop on Predictor Models in Software Engineering*, St. Louis, Missouri, pp. 1-5.

[60] G. Calikli, A. Tosun, A. Bener, M. Celik, The Effect of Granularity Level on Software Defect Prediction, *2009 24th International Symposium on Computer and Information Sciences*, Guzelyurt, Northern Cyprus, 2009, pp. 531-536.

[61] Y. Kamei, E. Shihab, Defect Prediction: Accomplishments and Future Challenges, *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 5*, Osaka, Japan, 2016, pp. 33-45.

[62] C. Pan, M. Lu, B. Xu, H. Gao, An Improved CNN Model for Within-Project Software Defect Prediction, *Applied Sciences*, Vol. 9, No. 10, Article No. 2138, May, 2019.

[63] J. Liu, J. Ai, M. Lu, J. Wang, H. Shi, Semantic Feature Learning for Software Defect Prediction from Source Code and External Knowledge, *Journal of Systems and Software*, Vol. 204, Article No. 111753, October, 2023.

[64] N. A. A. Khleel, K. Nehéz, A Novel Approach for Software Defect Prediction Using CNN and GRU Based on SMOTE Tomek Method, *Journal of Intelligent Information Systems*, Vol. 60, No. 3, pp. 673-707, June, 2023.

[65] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan, I. Zada, Software Defect Prediction Employing BiLSTM and

BERT-Based Semantic Feature, *Soft Computing*, Vol. 26, No. 16, pp. 7877-7891, August, 2022.

[66] Y. Sun, X. Y. Jiang, F. Wu, J. Li, D. Xing, H. Chen, Y. Sun, Adversarial Learning for Cross-Project Semi-Supervised Defect Prediction, *IEEE Access*, Vol. 8, pp. 32674-32687, February, 2020.

[67] L. Zhao, Z. Shang, L. Zhao, A. Qin, Y. Y. Tang, Siamese Dense Neural Network for Software Defect Prediction with Small Data, *IEEE Access*, Vol. 7, pp. 7663-7677, December, 2018.

[68] J. Xu, F. Wang, J. Ai, Defect Prediction with Semantics and Context Features of Codes Based on Graph Representation Learning, *IEEE Transactions on Reliability*, Vol. 70, No. 2, pp. 613-625, June, 2021.

[69] J. Liu, J. Lei, Z. Liao, J. He, Software Defect Prediction Model Based on Improved Twin Support Vector Machines, *Soft Computing*, Vol. 27, No. 21, pp. 16101-16110, November, 2023.

[70] S. Albahli, G. N. A. H. Yar, Defect Prediction Using Akaike and Bayesian Information Criterion, *Computer Systems Science and Engineering*, Vol. 41, No. 3, pp. 1117-1127, November, 2022.

[71] S. Goyal, Handling Class-Imbalance with KNN (Neighbourhood) Under-sampling for Software Defect Prediction, *Artificial Intelligence Review*, Vol. 55, No. 3, pp. 2023-2064, March, 2022.

[72] A. Khalid, G. Badshah, N. Ayub, M. Shiraz, M. Ghouse, Software Defect Prediction Analysis Using Machine Learning Techniques, *Sustainability*, Vol. 15, No. 6, Article No. 5517, March, 2023.

[73] S. Goyal, Genetic Evolution-Based Feature Selection for Software Defect Prediction Using SVMs, *Journal of Circuits, Systems and Computers*, Vol. 31, No. 11, Article No. 2250161, July, 2022.

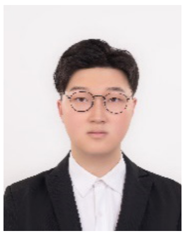
[74] J. M. González-Barahona, G. Robles, On the Reproducibility of Empirical Software Engineering Studies Based on Data Retrieved from Development Repositories, *Empirical Software Engineering*, Vol. 17, No. 1-2, pp. 75-89, February, 2012.

[75] W. Liu, B. Wang, W. Wang, Deep Learning Software Defect Prediction Methods for Cloud Environments Research, *Scientific Programming*, Vol. 2021, pp. 1-11. 2021.

[76] T. Lewowski, L. Madeyski, How Far Are We from Reproducible Research on Code Smell Detection? A Systematic Literature Review, *Information and Software Technology*, Vol. 144, Article No. 106783, April, 2022.

[77] C. Liu, C. Gao, X. Xia, D. Lo, J. Grundy, X. Yang, On the Reproducibility and Replicability of Deep Learning in Software Engineering, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 31, No. 1, pp. 1-46, January, 2022.

Biographies



Changjian Li is a graduate student in computer science at China University of Geosciences (Wuhan). His research interests include intelligent algorithms and software defect prediction.

Dongcheng Li earned his Ph.D. and M.S. degrees in Software Engineering from the University of Texas at Dallas and holds a B.S. in Computer Science from the University of Illinois, Springfield. Currently, he serves as an Assistant Professor in the Department of Computer Science at California State Polytechnic University, Humboldt. His research is centered on search-based software engineering, test generation, program repair, and intelligent optimization algorithms.

Hui Li is a professor at China University of Geosciences (Wuhan). Her main research direction is to use intelligent computing theories and methods to solve various complex problems in the aerospace field.

W. Eric Wong received his Ph.D. in computer science from Purdue University. He was at Telcordia Technologies (formerly, Bellcore) as a Senior Research Scientist and a Project Manager, where he was in charge of dependable telecom software development. He is currently a Full Professor and the Founding Director of the Advanced Research Center for Software Testing and Quality Assurance, Computer Science Department, The University of Texas at Dallas. He also has an appointment as a Guest Researcher with the National Institute of Standards and Technology (NIST). His research focuses on software testing, debugging, risk analysis/metrics, safety, and reliability. In 2014, he was named as the IEEE Reliability Society Engineer of the Year.

Man Zhao is currently an associate professor at China University of Geosciences (Wuhan). Her main research directions are computer science and technology, intelligent computing and artificial intelligence.