Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Survey paper

A systematic literature review on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools

Jalaj Pachouly [a], Swati Ahirrao [a], Ketan Kotecha [b,*], Ganeshsree Selvachandran [c,*], Ajith Abraham [d]

[a] *Symbiosis Institute of Technology, Symbiosis International (Deemed University), Pune 412115, Maharashtra, India*
[b] *Symbiosis Centre for Applied Artificial Intelligence, Symbiosis International (Deemed University), Pune 412115, Maharashtra, India*
[c] *Department of Actuarial Science and Applied Statistics, Faculty of Business and Management, UCSI University, Jalan Menara Gading, 56000 Cheras, Kuala Lumpur, Malaysia*
[d] *Machine Intelligence Research Labs, Auburn, WA 98071, USA*

ARTICLE INFO

Keywords:
Software defect prediction
Classification
Artificial intelligence
Machine learning

ABSTRACT

Delivering high-quality software products is a challenging task. It needs proper coordination from various teams in planning, execution, and testing. Many software products have high numbers of defects revealed in a production environment. Software failures are costly regarding money, time, and reputation for a business and even life-threatening if utilized in critical applications. Identifying and fixing software defects in the production system is costly, which could be a trivial task if detected before shipping the product. Binary classification is commonly used in existing software defect prediction studies. With the advancements in Artificial Intelligence techniques, there is a great potential to provide meaningful information to software development teams for producing quality software products. An extensive survey for Software Defect Prediction is necessary for exploring datasets, data validation methods, defect detection, and prediction approaches and tools. The survey infers standard datasets utilized in early studies lack adequate features and data validation techniques. According to the finding of the literature survey, the standard datasets has few labels, resulting in insufficient details regarding defects. Systematic Literature Reviews (SLR) on Software Defect Prediction are limited. Hence this SLR presents a comprehensive analysis of defect datasets, dataset validation, detection, prediction approaches, and tools for Software Defect Prediction. The survey exhibits the futuristic recommendations that will allow researchers to develop a tool for Software Defect Prediction. The survey introduces the architecture for developing a software prediction dataset with adequate features and statistical data validation techniques for multi-label classification for software defects.

1. Introduction

Producing reliable quality software products is a complex process that requires various team's combined effort for planning, execution, and testing. Many software products get high numbers of defects in the testing phase and even post-deployment in the production environment. As per the report (Anon, 2018b) published in September 2018, in the Consortium for IT Software Quality (CISQ), more than 50% of the total software cost is consumed in identifying and fixing defects and the losses due to software failures in a production environment. However, the total cost for software delivery is not entirely visible and can be correlated with the Iceberg model (Anon, 2018b), as shown in Fig. 1.

Another white paper, Software fails watch (5th edition) (Anon, 2018a), published in the Tricentis, analyzed the 606 well-known software failures. According to the article, 3.7 billion people were impacted, costing $1.7 trillion for assets and involving 314 companies. These are just a few numbers. Countless incidents have shown that poor-quality software can have an adverse impact on the business. In a few cases, it can be life-threatening if the software is used in life-critical systems. Another study carried out by Cambridge University's Judge Business School (Anon, 2013) stated that 30% to 50% of the cost is spent finding and fixing defects, implying that roughly half of the time spent by software developers is spent finding and fixing bugs. Identifying and correcting software defects is crucial for delivering a reliable software product with fewer bugs. Time and cost can be saved if the defects are caught in the early stage of the development, but it is

* Corresponding authors.
E-mail addresses: jalaj.pachouly.phd2019@sitpune.edu.in (J. Pachouly), swatia@sitpune.edu.in (S. Ahirrao), director@sitpune.edu.in (K. Kotecha), Ganeshsree@ucsiuniversity.edu.my (G. Selvachandran), ajith.abraham@ieee.org (A. Abraham).

https://doi.org/10.1016/j.engappai.2022.104773
Received 27 November 2021; Received in revised form 1 February 2022; Accepted 14 February 2022
Available online xxxx

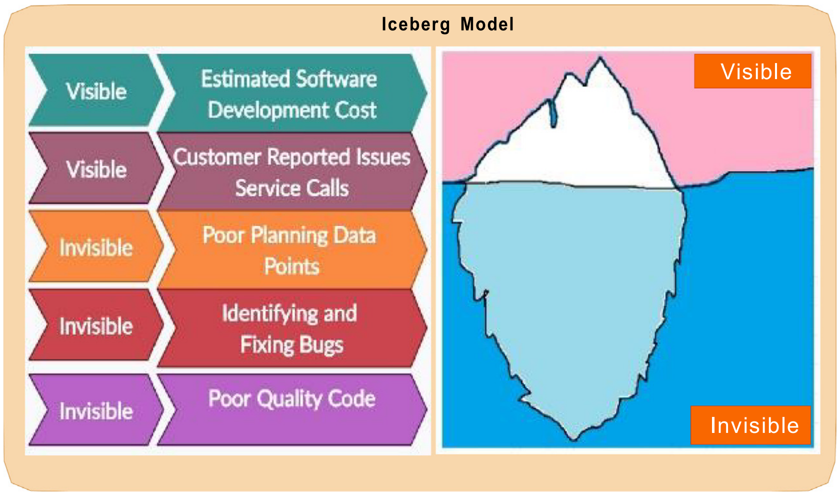
Iceberg Model
Visible
Estimated Software Development Cost
Visible
Customer Reported Issues Service Calls
Invisible
Poor Planning Data Points
Invisible
Identifying and Fixing Bugs
Invisible
Poor Quality Code
Visible
Invisible

Fig. 1. Software development cost — Iceberg model (Anon, 2018b).

most costly if the defects are found later. It also causes damage to the organizations financially or in terms of data loss or user inconvenience. There is a pressing need to build a system that improves the complete software delivery process to reduce the cost and time required to produce high-quality software. Software quality and timely delivery of the software products are the number one priority for a successful, profitable business. In the direction of improvement, a lot can be learned from the historical data. A considerable amount of information is continuously captured in the various digital platforms used in the software development phase, like planning, development, and testing. Many companies use Jira, Confluence, Jenkins for Continuous Integration, Perforce, and GitHub for Code Version Management. Such tools capture the relevant artifacts during the Software Development Life Cycle. Considering that we have tons of data available, intelligence can be generated using relevant data and technology. Predicted Intelligence can be instrumental in enhancing the reliability of the overall software delivery process. It can help decrease the rework cost and have more accurate estimates to meet the estimated deadlines with fewer defects.

1.1. Outline of the paper

Fig. 2 shows the outline of this Systematic Literature Review.

The paper starts with the introduction section, which contains four subsections detailing the significance, evolution timeline, motivation for writing this paper, and background of defect identification and the prediction. The background section provides common defect identification and prediction practices using manual testing, automation testing, and prediction approaches. The prior research section discusses a few significant studies on Software Defect Prediction and states the research goals and the study's contribution. The terminology section briefly describes commonly used terms in Software Defect Prediction. The research methodology section describes the SLR process. We followed the Kitchenham (Kitchenham et al., 2009) guidelines. The research methodology section contains the keywords used for searching the research studies in various databases. Selection criteria talk about how the papers are selected for this SLR. Inclusion and Exclusion criteria discuss the criteria used to select and filter the papers. The selection of the paper is made based on the various parameters and the score count of the paper. The literature outcome section describes the analysis of the previous studies against the formulated research questions. The discussion section summarizes the literature outcome on formulated research questions. The limitation of the study section discusses possible limitations present in this SLR. The conclusion section concludes the findings as per the analysis. Finally, the future work and opportunity section discusses the proposed research work based on the limitation found in the earlier research.

1.2. Significance and relevance

Delivering software products with the high-quality demands that delivered software is bug-free and performs as expected in the production environment. The number of defects can be predicted early, before delivering the product. Most defects can be found and fixed optimally to deliver the product on the decided timeline using Artificial Intelligence techniques. Manual testing and automation testing are the traditional approaches that execute a well-defined, limited number of test cases. Due to limited resources like human resources and time, traditional defect finding techniques can detect fewer defects. It also does not leverage historical defects encountered in the production system for similar projects or earlier product versions. Getting more probable defects based on the historical dataset can significantly make the product more robust. Artificial Intelligence techniques are very useful, especially Machine Learning and Deep Learning, due to their vital role in predicting software defects due to great classification capability. Artificial Intelligence approaches can be used to forecast additional essential information such as defect severity, defect estimations, code references, resource allocation, and defect types in addition to software flaws. Any software development team may use this information to plan their future development effort based on real historical data and help them make the best decisions possible.

1.3. Evolution timeline

Prediction of software defects is explored as an active research field. Fig. 3 shows the evolution of the Software Defect Prediction over the decade.

In this evolution diagram, learning algorithms are classified by commonality in their operation and with usages like CPDP, WPDP, CVDP, and HDP. Table 22, in the miscellaneous section, contains more details of algorithms that are grouped based on similarity. The evolution history of Software Defect Prediction is shown in the four years of the span, starting with 2010.

2010 to 2013: Initial studies of the Software Defect Prediction were focused on evaluating various Machine Learning algorithms for improving the accuracy of the prediction and the performance. Naive Bayes, Bayesian Belief Network (BBN), and Bayesian Network are based on Bayesian Algorithms. Support Vector Machine (SVM) is an instance-based Machine Learning approach that uses similarity measurements to predict new software defects. K-means, Hierarchical Clustering is the technique used to organize the data into groups with the greatest commonality to predict the new defect. Adaboost, Boosting is the Ensemble Algorithms used to enhance the prediction using a combination

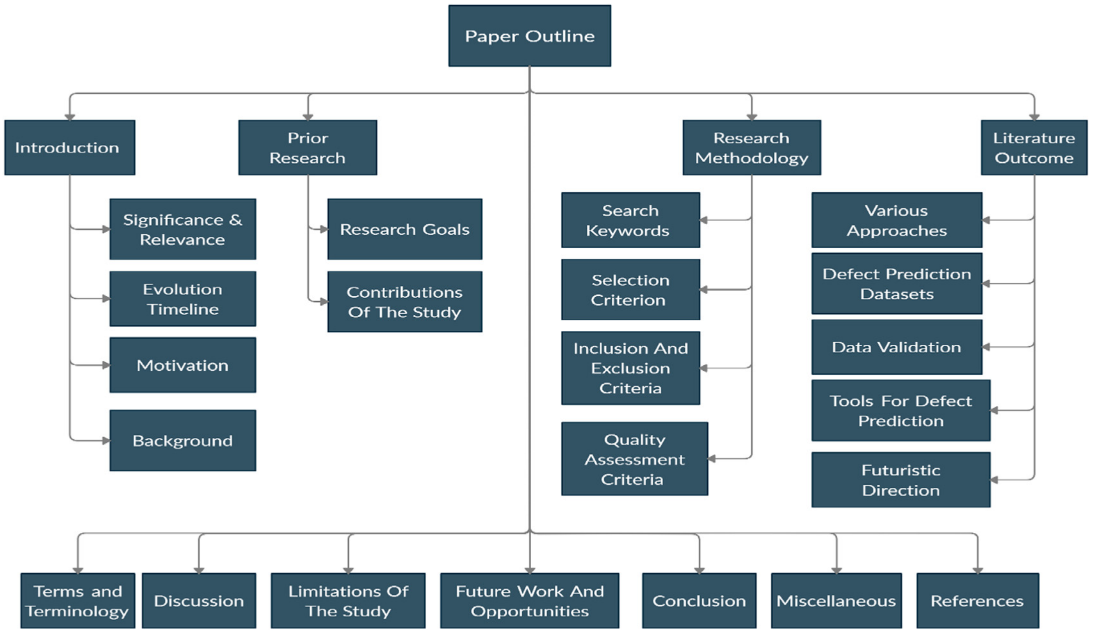

Fig. 2. Outline of the Systematic Literature Review.

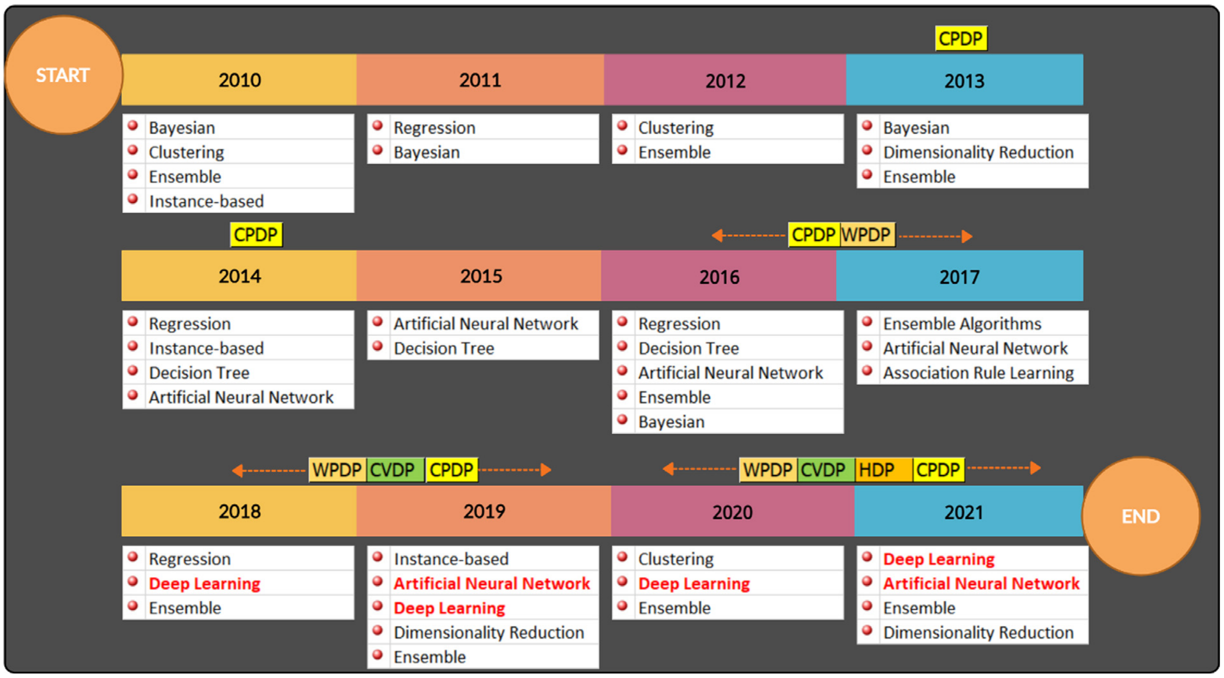

Fig. 3. Evolution timeline for Software Defect Prediction [2010–2021].

of many weaker models. Finally, Linear Regression uses a linear model for prediction based on the training data.

2014 to 2017: Aside from existing Machine Learning techniques, two new ones gained traction in 2014, namely Artificial Neural Networks and Decision Tree-based algorithms. Radial Basis Function, Back Propagation (BPA) algorithm, Resilient Back Propagation, Stochastic Belief Network, Label Propagation are some of the references of Neural Networks Algorithms used. Logistic regression and Multiple Linear Regression are used, which are Regression-based algorithms. K-Nearest Neighbor regression is used to support nonlinear model data. A Bayesian Regularization algorithm reduces squared errors and weights, determining the best combination to form an efficient network. C4.5, Decision Forest, CART are some of the Decision Tree algorithms used. Ensemble Learning-based techniques include Gradient Boosting, Real Adaboost, and Stacking. A subset of Association Rules is used to explore the relation between features and classes to increase prediction accuracy. Rules are used to study the link between features and classes to improve prediction accuracy.

2018 to 2021: It is clear from the SLR that Deep Learning, Artificial Neural Networks started great attraction in the last four years. Deep belief networks, Convolutional Neural Networks (CNNs), Tree-based CNN, Deep CNN, Recurrent Neural Networks (RNN), Layered Recurrent Neural Network, Long Short Term Memory Network, Bi-directional Long Short Term Memory, Autoencoder, Variational Autoencoder, and Staked denoising auto-encoder are just a few Deep Learning techniques that have been used in recent years. Furthermore, Artificial Neural

Networks such as Multi-Layer Perceptron are used for accurate defect prediction. Ridge regression (RR), Lasso regression (LR), Elastic-net regression are the new addition in the Regression category. Over-bagging, Subbagging, Isolation Forest, Deep Forest, and TrAdaBoost are more variations in Ensemble Learning. Artificial intelligence techniques, particularly Deep Learning and Artificial Neural Networks, have received much interest in the last four years for Software Defect Prediction, as seen in the evolution diagram. Many researchers did their study to predict the defects using several Artificial Intelligence techniques. The evolution timeline shows that early research, up until 2016, was primarily focused on Cross Project Defect Prediction. Within Project Defect Prediction has received more interest since 2016. In the domain of Software Defect Prediction, Cross Version Defect Prediction and Heterogeneous Defect Prediction are relatively new trends.

1.4. Motivation

The existing literature is lacking on systematic literature reviews for Software Defect Predictions. Earlier research is mainly focused on defect classification using publicly available datasets. Prediction outcome is limited, and there are no actionable items for the software development team. There is a lack of a survey that focuses on the legacy versus modern approaches for identifying software defects. Earlier surveys did not cover the existing tools. Before using the data to train the classifiers, the adequate focus was not given to the data validation techniques. Hence, a comprehensive survey needs to focus on datasets, data validation methods, defect detection and prediction approaches, tools, and recommendations for further research. This Systematic Literature Review focuses on datasets, methods for data validation, defect detection and prediction approaches, tools, and recommendations for future researchers. According to a bibliometric review (Pachouly et al., 2020), there is a lot of interest in the field of Software Defect Prediction among researchers aworldwide

1.5. Background

Identifying and fixing software defects is an important task to ensure quality products. The software development process gets initiated once the customer has a business requirement. The development process involves gathering requirements, feasibility studies, and generating high-level design documents. The enormous task of developing software gets broken into multiple small tasks like creating numerous use cases. At the same time, the Quality Assurance engineer or the product owner comes up with the test case design to ensure that the software development must meet the business requirements. Quality Assurance engineers make a suite of test cases that can be manually executed or automated for execution. Once the code is ready for testing, it is tested by the testing team, and the testing team comes up with the test reports. A few failed test cases report a bug in the test report, which means the business requirement is not met as expected. Bugs are reported when the test engineer cannot execute or complete the business flow due to other technical issues. As per Fig. 4, manual and automation testing finds defects and logs into the defect dataset.

Few test cases that meet the business requirements are marked as passed. The Quality Assurance engineer logs the defect information into the defect dataset using the defect tracking system, which creates the defect dataset for one specific software product release. We can use this defect dataset and the software repository to develop the defect prediction dataset. Defect dataset can check if the software code can be labeled as Buggy or Non-Buggy? Usually, the specific source file is marked as Buggy if a bug is reported for that particular code. The next step is to identify the relevant features for the defect prediction once we have the defect prediction dataset available with the right attributes. Samples are determined using various sampling techniques, and the dataset can be validated to see if the data quality is good enough to predict software defects. Fig. 5 shows the general approach used for Software Defect Prediction.

2. Prior research

The Systematic Literature Review (SLR) aims to answer the formulated research questions by critically investigating the existing research papers for predicting software defects. According to a literature search, only a few studies have conducted an SLR for Software Defect Prediction. As per our findings, four survey papers explored defect predictions. This section discusses previous studies in the field of Software Defect Prediction that conducted an SLR. Catal et al. (Catal and Diri, 2009) reviewed the 74 articles collected between 1990–2007. The survey was focused on datasets, methods, and metrics. One of the limitations reported in the survey is that many researchers have used private data source that is inaccessible and leads to non-reproducible research. As per the survey, Machine Learning techniques are dominant in the field of Software Defect Prediction. Class, method, file, component process, and quantitative levels are the six types of metrics. Machine Learning and Statistical approaches are combined for defect predictions. As per the survey, significant algorithms for solving defect prediction problems are Naive Bayes, Random Forests, and J48. For feature reduction, Principal Component Analysis is found significant. One of the limitations of the survey was referring to the smaller number of papers. The survey recommends using Machine Learning models rather than Statistical methods, as prediction results are better with Machine Learning. The survey also recommends using public datasets, as public datasets are accessible, and future researchers can use them to improve their research by employing repeatable and additional verification methods. Hall et al. (Hall et al., 2011) focused on identifying the impact on the fault prediction performance with the independent variable, the context of the model, and chosen methods. Two hundred eight studies between the years 2000 to 2010 are selected for the analysis. The purpose of the survey was to provide guidance and the path for the future researcher to choose the appropriate model based on the context of their study. The study also found that the model performs well with more data, which means the size influences the model's performance. As per the survey, the simple model performs well using a simple Naive Bayes and Logistic Regression technique. In contrast, SVM seems to underperform as it needs parameter optimization, which is not considered in many research studies. Process-related metrics do not outperform Object-Oriented metrics or Source Code metrics, according to a survey based on 19 studies. However, performance does not change if we choose only LOC or use object-oriented metrics. Nineteen conducted studies suggest that if we pick independent variables and use a combination of the few metrics set, it helps to improve the prediction performance. Survey also analyzed the prediction performance on data quality. It is tough to gather high-quality data, and only a few studies have examined data validation. Handling class imbalance is significant for the quality of the fault prediction. The research confirmed the effect of the class imbalance on the quality of prediction results. The survey indicates that one research study indicated the severity of the defects as medium and high, although the severity is subjective and difficult to generalize. Still, severity is important information that helps the development team prioritize and focus on the high severity issues. Sobrinho et al. (Hall et al., 2011) survey various substandard code practices, making the products inferior. This survey does not discuss the prediction of the defect directly. However, it gives useful information about the bugs and the quality issues in the code that any software development team needs to know to produce a quality product. Some of the bad smells are duplicate code, anti-pattern, and BLOB class or GOD class which is well known to produce negative consequences. GOD class is identified based on the number of instance variables, or in other words, class, which is doing many things and does not have a single atomic goal. GOD class is a code smell that makes maintenance of the code hard and produces more defects associated with such classes. Sometimes such challenges are referred to as Technical Debt. The more the technical debt, the less the product quality and the huge possibility of the unidentified hidden defects in the delivered

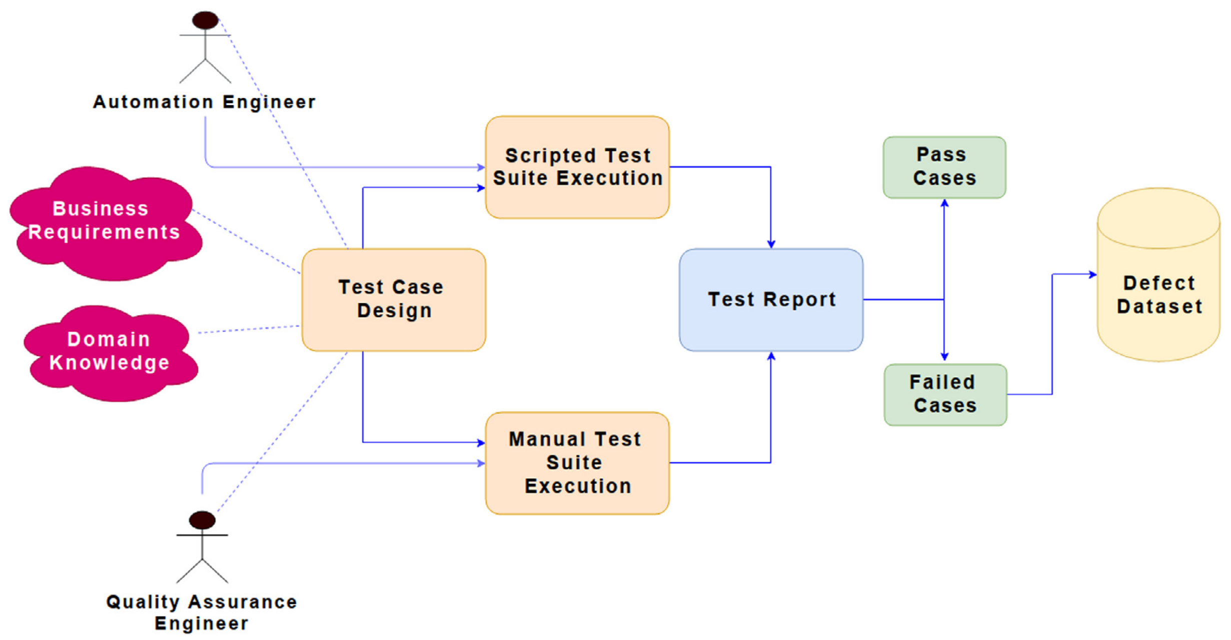

Fig. 4. Manual and automation testing approach.

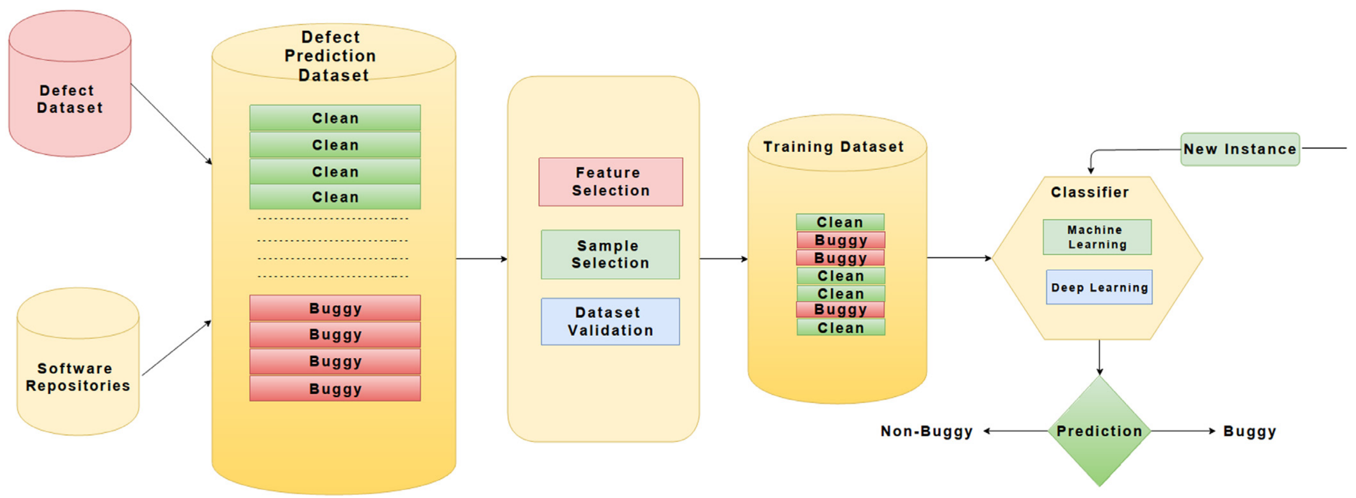

Fig. 5. Software Defect Prediction — general approach.

software. The survey's focus was analyzing the source code with low-level structure and identifying the bad coding architectural decisions that make it sub-optimal and of low quality. The survey also identified that the development community started paying attention in recent years to keep track of the bad smell in the code and keep it as low as possible. Hosseini et al. (Hosseini et al., 2017) worked to better understand the state of the art in CPDP in terms of metrics, models, data techniques, datasets, and associated performance. They also compared the performance of CPDP vs. WPDP models. Nearest-Neighbor and Decision Tree models perform well in CPDP. However, the popular Naive Bayes models get average results. Ensemble performance varies substantially depending on the F-Measure and AUC. For data strategies that handle CPDP problems, row/column processing enhances Recall for CPDP, although it lowers precision. This has been seen on several occasions, notably in a meta-analysis comparing CPDP with WPDP. The records from NASA and Jureczko appear to favor CPDP over WPDP more frequently. CPDP is still a problem, and additional study is needed before reliable applications can be implemented. Table 1 provides a summary of existing SLR investigations. Prior research surveys mentioned above address various questions related to identifying metrics, figuring out the classification algorithms, predicting performance, and exploring the impact of the bad smell on the code. Although as per the observation, the number of the surveys done on the Software Defect Prediction is very low, and here are the few limitations observed in the existing surveys:

1. Most early research was merely involved in the code classification as Buggy vs. Non-Buggy. The survey was limited to a comprehensive analysis of the binary classification, which is useful information but does not provide any futuristic direction.
2. Existing surveys did not generate actionable task items, which the development teams could leverage. Existing literature did not explore additional information related to defects, like what code changes are required to fix the defects, the estimate for the defects, and who is the right resource to fix the defect?
3. Most of the research has used the publicly available dataset in the existing literature, which was created for defect classification only. Hence, it did not contain the adequate feature to generate additional useful information for the defects.

Table 1
A summary of existing SLR.

Sr. No.	SLR	Year	Focus	Key Observations
1	A systematic review of software fault prediction studies (Catal and Diri, 2009)	2009	The survey's focus was metrics, methods, and datasets used around Software Defect Prediction.	ML usage increased from 2005. Due to the usage of private data, many pieces of research are not repeatable. Method Level Metrics are most dominant for defect prediction. They recommended the usage of Class level metrics in the Design phase of Software development.
2	A Systematic Literature Review and Meta-analysis on Cross Project Defect Prediction (Hosseini et al., 2017)	2017	Gain a better understanding of the state-of-the-art for CPDP in terms of models, metrics, datasets, techniques, and associated performances.	Nearest-Neighbor and Decision Tree models perform well for CPDP; however, the popular Naive Bayes models get average results. Ensemble performance varies substantially depending on the F-measure and AUC. The records from NASA and Jureczko appear to favor CPDP over WPDP.
3	A systematic literature review on bad smells –5 W's: which, when, what, who, where (Sobrinho et al., 2018)	2018	The review's focus was to investigate the bad smells in the code and the work done by early researchers. What kind of bad smells in the code is causing the issues? The findings also point to future efforts against the bad smell.	Duplicate code bed smell is studied well and classified as Duplicate Code Group (DCG). Other smells are grouped as Other Bed Smell Group (OSBG). 69.8% of research is conducted around the Duplicate Code Group.
4	A Systematic Literature Review on Fault Prediction Performance in Software Engineering (Hall et al., 2011)	2011	The study looked at how the context of models, the independent variables employed, and the modeling methodologies used influenced the performance of defect prediction models.	Most research provides insufficient contextual and methodological information to fully comprehend a model. As a result, it is difficult for potential model users to find one that fits their needs, and just a few models have made it into industrial practice.

4. The existing survey did not pay attention to the challenges for collecting the quality dataset for the Software Defect Prediction. Collecting the quality data with the relevant feature is difficult. Creating or extending existing datasets is necessary if we extend the prediction to have more useful information than binary classification.
5. The prior survey did not analyze the existing tools available at our disposal for the Software Defect Prediction.
6. The existing survey did not pay enough attention to the dataset validation, which is very important and ensures the capability of the training model to predict the defects.

It gives enough justification for conducting a Systematic Literature Review that will explore various prediction approaches, datasets, validation techniques, existing tools, and possible futuristic recommendations, which can be helpful for new researchers. Historical defect datasets contain useful information that can give the software development team significant pointers. Hence Systematic Literature Review is required exploring on the dataset challenges, appropriateness of the dataset with the set of relevant features, existing available tools and, extending the scope of prediction with additional useful information around the defect. This SLR presents the comprehensive analysis of defect-finding approaches, data validation methods, existing tools available for Software Defect Prediction, and Artificial Intelligence techniques to bridge the gap and make the prediction actionable. Such information will undoubtedly aid future researchers in gaining a thorough understanding of previous work and the obvious next step for predictions related to software defects.

2.1. Research goals

The major purpose of SLR is to conduct a critical study of available approaches in the context of Software Defect Prediction to discover solutions to the research question posed in Table 2.

2.2. Contributions of the study

Contributions made by Systematic Literature Review are as given below:

1. A comprehensive analysis is done for popular approaches (detection and prediction based) employed for software defect identification.
2. A comprehensive analysis was done on the available datasets and explored the challenges faced in the publicly available Software Defect Prediction datasets like Class Imbalance, Feature Selection, Sampling requirements.
3. Explored the dataset validation methods used in the early research and identified that most research studies lack data validation methods.
4. For Software Defect Prediction, a comprehensive analysis of Artificial Intelligence techniques, notably Machine Learning and Deep Learning approach, is presented.
5. Summarized the available existing tools and the frameworks, which are specifically designed for Software Defect Prediction.
6. Proposing Multi-Label classification for extending the scope of defect prediction for defect severity, defect estimates, code references, resource allocation, and defect types using Artificial Intelligence techniques and ensuring a custom dataset has the adequate features for such prediction.
7. The study is proposing the Architecture that creates the balanced dataset from the Open-Source Repository (GitHub) with adequate features for Multi-Label classification and gets more significant information about predicting and fixing the software defects. The architecture also focuses on the dataset validation methods to ensure a Better-quality dataset that predicts Software Defect Predictions.

3. Terminology

Fig. 6 shows various common terms related to software defects. The study mentions a detailed taxonomy of Software Defect Predictions (Caulo and Scanniello, 2020).

3.1. Software defect

A software defect is a malfunctioning of the software due to which the end-user requirement does not get fulfilled.

3.2. Defect prediction

Defect prediction is a mechanism that indicates possible defects in the newly written code or modified existing code without testing the code.

3.3. Agile methodology

A method of project management that divides a project into numerous phases.

Table 2
Research questions.

Number	Research Questions	Discussion
RQ1	What are the various approaches for finding defects in the newly developed code?	Do we need to investigate the various defect-finding approaches used to identify software defects? What are the present trends and prospects for detecting software defects?
RQ2	What are the available datasets, and are they good enough for predicting various actionable tasks related to defects?	Merely classifying the code as Buggy or Not-Buggy is not adding much value to the development team. Previous studies have concentrated on datasets with limited binary classification capabilities. We need to explore if the existing datasets are good enough to predict code pointers, resource allocation, and estimation, among others.
RQ3	What are the available data validation techniques to ensure that training data is appropriate for Software Defect Prediction modeling?	The exploration of Parametric and Non-Parametric tests to validate the appropriateness of the dataset for the desired prediction around the software defects.
RQ4	What are the various tools/frameworks available for Software Defect Prediction?	Do we have any existing tools commercially available for defect-related predictions? What are the advantages and the shortcomings?
RQ5	What is the possible futuristic direction for Software Defect Prediction, and what can be predicted for software defects to enhance the standard of the delivered software product?	This research question seeks a futuristic framework or tools based on Artificial Intelligence techniques that can predict additional information about software defects, such as resource allocation, defect estimates, and possible code fixes, to enhance the quality of the produced source code.

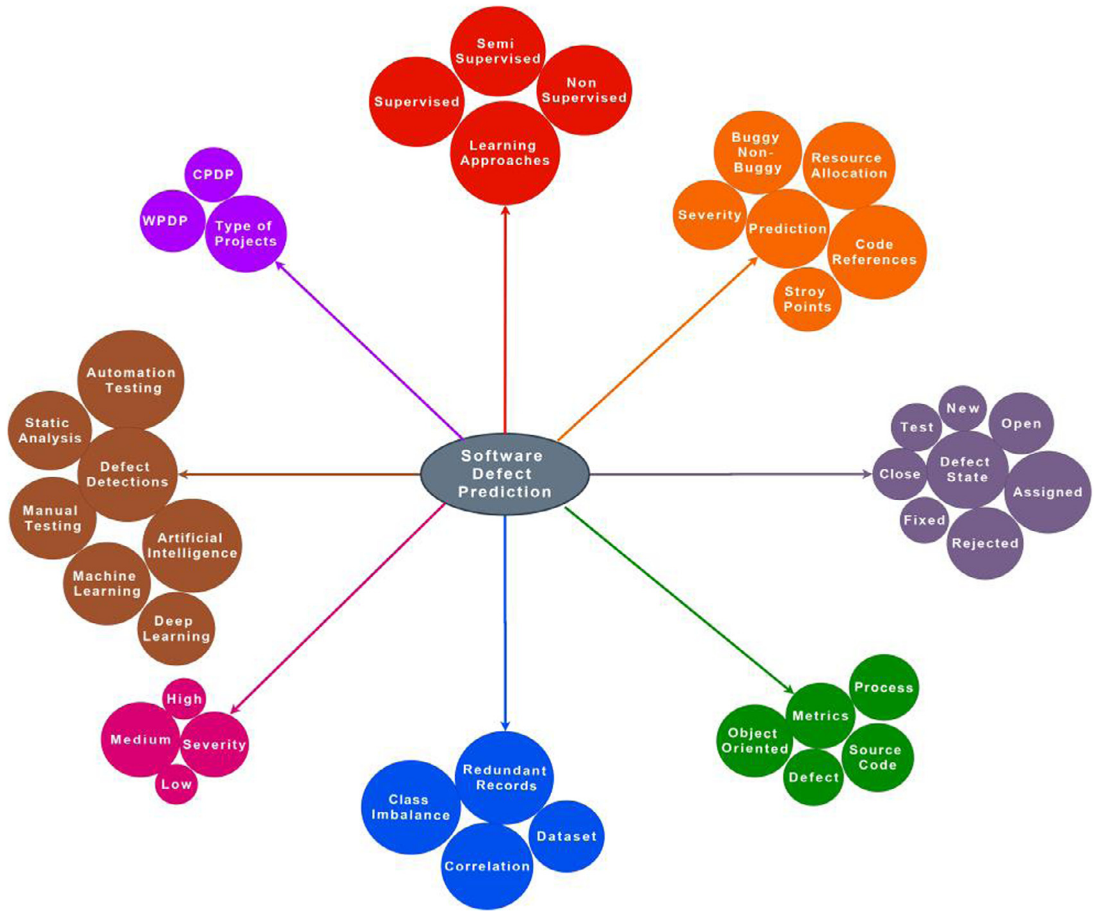

Fig. 6. Software Defect Prediction—Terminology..

3.4. Confluence

Confluence is a collaborative workspace for teams that combines knowledge and collaboration.

3.5. Resource allocation

Resource allocation refers to allocating the human resources for fixing the defects. Resource allocations involve combining the tester and developer.

3.6. Story point

Story point is an estimation technique mostly used in the Agile-based development system, where the story point is given using the Fibonacci series. The valid numbers sequence can be — 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.

3.7. Code references

Here, code references highlight probable code modifications that developers should fix the defects. One way to enhance efficiency is

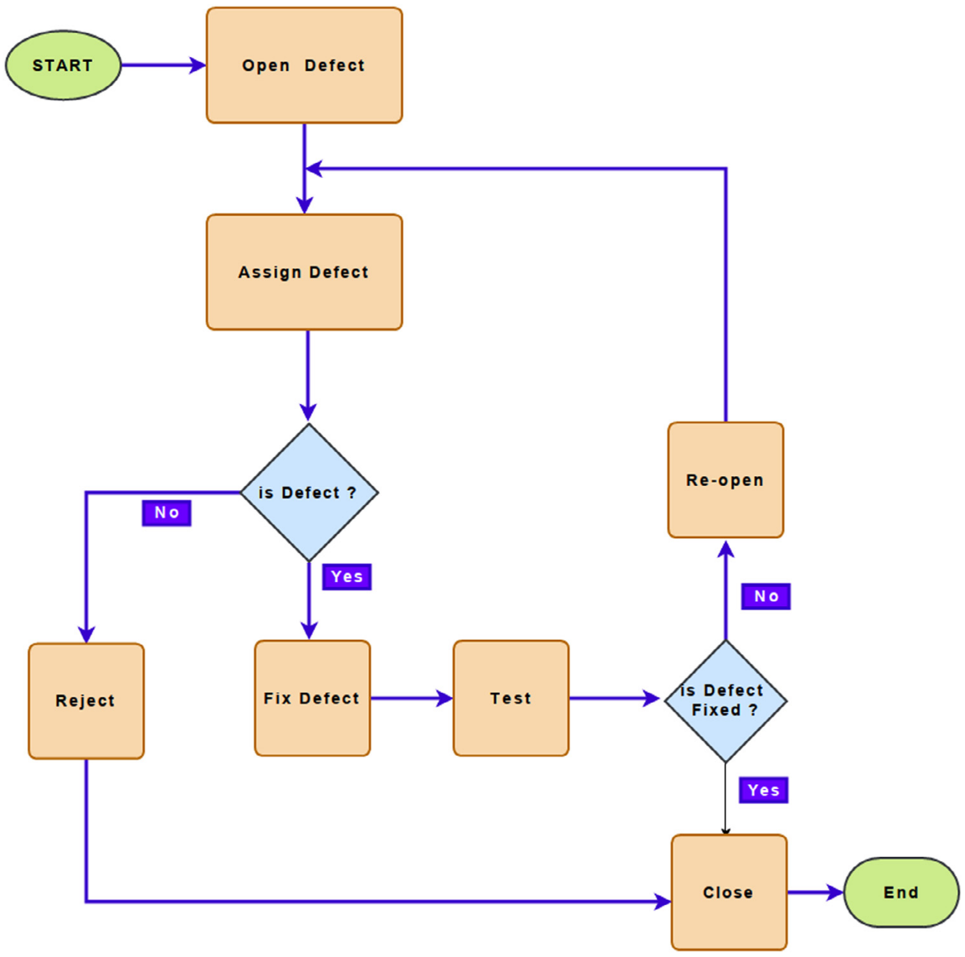

Fig. 7. Defect life cycle.

to give software developers clues to potential areas of the code that need to be changed to fix either the predicted or detected defects.

3.8. *Defect life cycle*

The defect life cycle of the defect is shown in Fig. 7, indicating the transitions of defect states.

3.9. *Defect states*

The defect discovered during testing can be in a variety of states. Table 3 shows various states of the defect.

4. Research methodology

Formulating the right research question is crucial for any SLR. These research questions are framed using the PIOC (Population, Intervention, Outcome, Context) approach published by Kitchenham (Kitchenham et al., 2009), as shown in Table 4. PRISMA guidelines published by Kitchenham and Charters (Kitchenham et al., 2009) are referred into the presented SLR to answer the formulated research questions.

4.1. *Search keywords*

Search keywords used for the Systematic Literature Review are Software Defect Prediction, Software Fault Prediction, Software Bug Prediction, Software Defect Forecasting, Artificial Intelligence, and Machine Learning. Fig. 8 shows the results from the database searches and filtering of papers on various criteria. Fig. 9 shows the SLR process.

4.2. *Selection criterion*

Table 5 shows search keywords and queries used as a search strategy for selecting the data for this SLR. In the presented paper, research is limited to 2010 to 2021.

4.3. *Inclusion and exclusion criteria*

Additional filter criteria are employed to find high-quality studies. Table 6 shows a few filter criteria applied to this SLR.

Table 7 displays the number of filtered studies based on the criteria

4.4. *Quality assessment criteria*

Quality assessment for the selected studies is driven by the guidelines provided by Kitchenham et al. (2009). These guidelines help filter out the articles missing the proof of the significant findings claimed in the paper, empirical analysis, justification of the proposed arguments, etc. The selection of an article is based on the score achieved, based on the guidelines. As per Fig. 10, the minimum qualification score for the selected study is 4.

5. Literature outcome

The Systematic Literature Review was conducted with 146 research studies in Software Defect Prediction to get details. This Section discusses various approaches used for defect identification. It includes defect identification based on defect detection and defects prediction. The distribution of topics across the 146 selected studies is shown in Fig. 11.

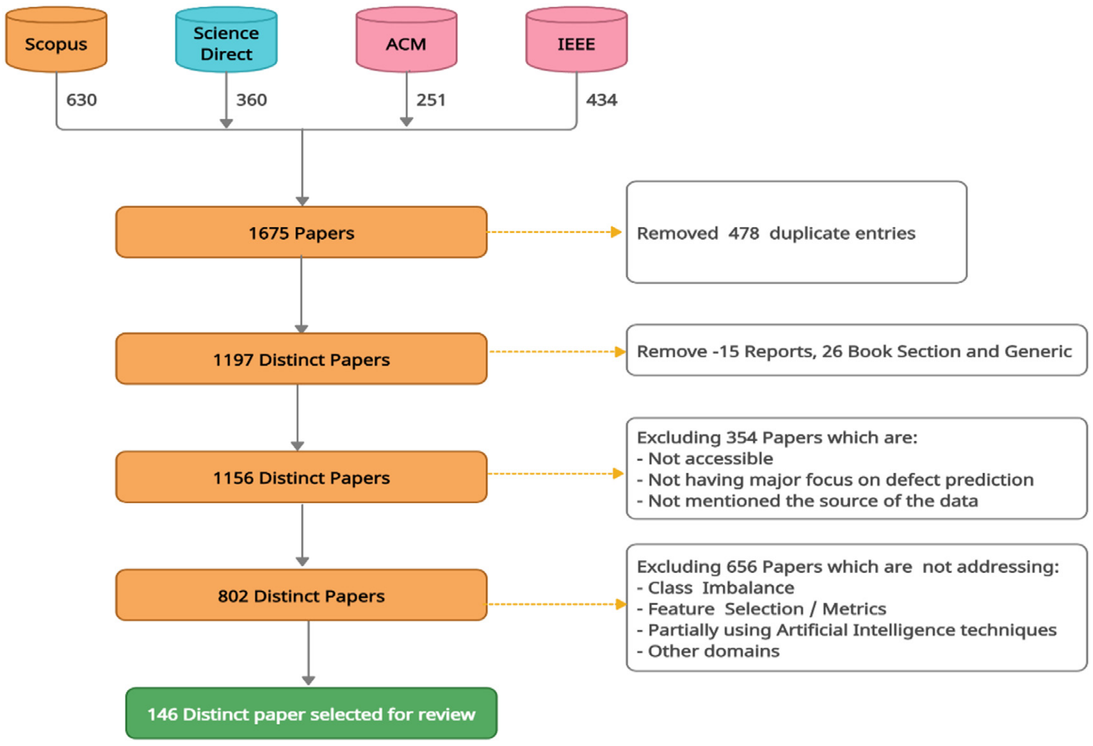
Scopus
Science Direct
ACM
IEEE
630
360
251
434
1675 Papers
Removed 478 duplicate entries
1197 Distinct Papers
Remove -15 Reports, 26 Book Section and Generic
1156 Distinct Papers
Excluding 354 Papers which are:
- Not accessible
- Not having major focus on defect prediction
- Not mentioned the source of the data
802 Distinct Papers
Excluding 656 Papers which are not addressing:
- Class Imbalance
- Feature Selection / Metrics
- Partially using Artificial Intelligence techniques
- Other domains
146 Distinct paper selected for review

Fig. 8. Database search results for Software Defect Prediction.

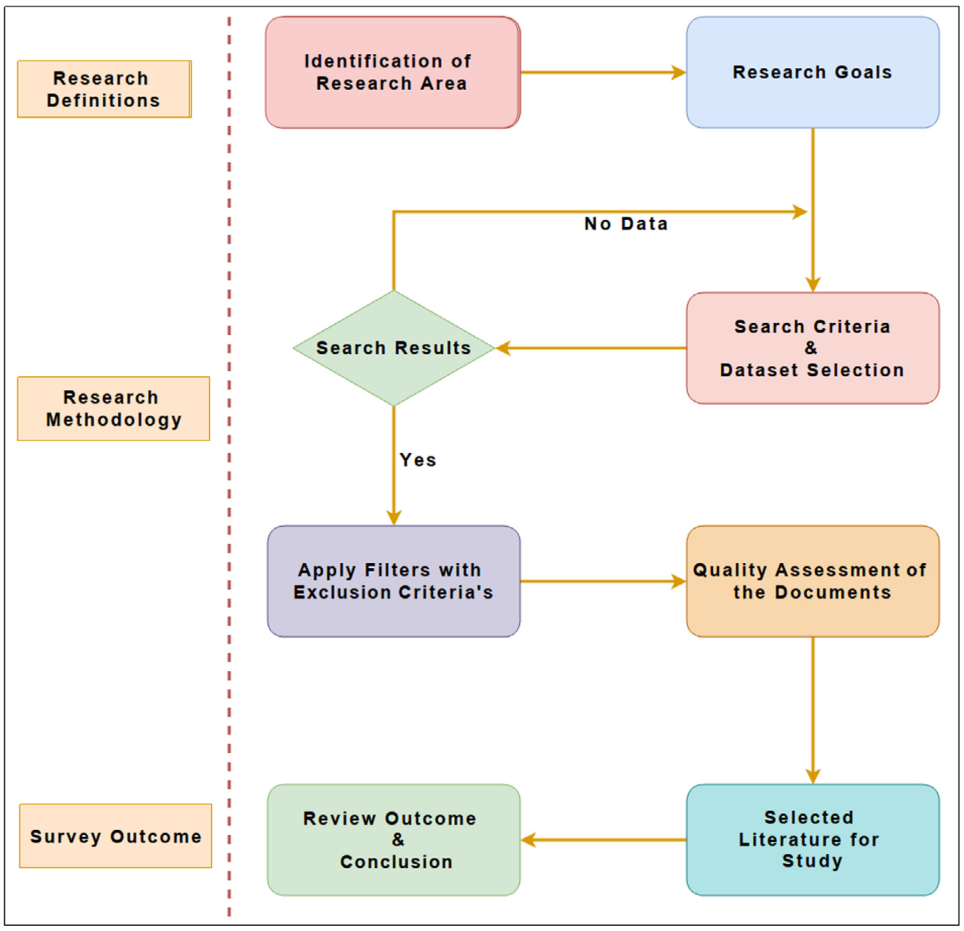
Research Definitions
Identification of Research Area
Research Goals
No Data
Research Methodology
Search Results
Search Criteria & Dataset Selection
Yes
Apply Filters with Exclusion Criteria's
Quality Assessment of the Documents
Survey Outcome
Review Outcome & Conclusion
Selected Literature for Study

Fig. 9. The SLR process.

Table 3
Defect states.

State	Description
New	When the defects first surfaced in the testing and were logged by the quality engineers as a defect.
Open	The defect is logged and acknowledged, although nobody has taken any action. It is not assigned to anyone and, so far, has no investigation on it.
Assigned	After the initial screening, the defect is assigned to the appropriate team for fixing.
Reject	The developer can reject the defect if it is as per the design.
Fixed	The developer makes the appropriate code or configuration changes, ensures the issue is resolved, and then marks the defect as fixed so that the testing team can verify the defect fix.
Test	The testing team picks the defect with status as fixed for the verification.
Re-open	If the testing team finds that the fix is not functioning as expected, they can re-open the defect by updating the comments and marking it as Opened.
Close	If the testing team finds that the code is working as expected, they can mark the defect as verified and close the defect.

Table 4
PIOC (Population, Intervention, Outcome, Context) criteria.

Parameter	Meaning	Keywords Used
Population	It is a field of application.	"Software defect prediction" OR "software fault prediction" OR "Software bug prediction" OR "software defect forecasting"
Intervention	It is a software methodology or approach for dealing with a certain problem.	"Artificial Intelligence" OR "Machine Learning"
Outcome	It should address issues that matter to practitioners, such as increased reliability, lower production costs, and shorter time to market.	"Reduce cost" OR "Reduce time to market."
Context	It is the setting in which the intervention takes place.	"Software Defect Prediction" OR "Code references"

Table 5
Search queries on various databases with parameters.

Scopus	(TITLE-ABS-KEY ("software defect prediction") OR TITLE-ABS-KEY ("software fault prediction") OR TITLE-ABS-KEY ("software bug prediction") OR TITLE-ABS-KEY ("software defect forecasting.")) AND (("artificial intelligence")) AND ("machine learning") AND (LIMIT-TO (PUBYEAR , 2010) OR LIMIT-TO (PUBYEAR , 2011) OR LIMIT-TO (PUBYEAR, 2012) OR LIMIT-TO (PUBYEAR, 2013) OR LIMIT-TO (PUBYEAR, 2014) OR LIMIT- TO (PUBYEAR, 2015) OR LIMIT-TO (PUBYEAR, 2016) OR LIMIT-TO (PUBYEAR, 2017) OR LIMIT-TO (PUBYEAR, 2018) OR LIMIT-TO (PUBYEAR, 2019) OR LIMIT-TO (PUBYEAR, 2020) OR LIMIT-TO (PUBYEAR, 2021)) AND (LIMIT-TO (LANGUAGE, "English"))	630
Science Direct	"software defect prediction" OR "software fault prediction" OR "soft-ware bug prediction" OR "software defect forecasting" AND "artificial intelligence" AND "machine learning" Year: 2010–2021 "	360
IEEE	("All Metadata": "software defect prediction") OR ("All Meta-data": "software fault prediction") OR ("All Metadata": "software bug prediction") OR ("All Metadata": " software defect forecasting") AND ("All Metadata": "artificial intelligence") AND ("All Meta-data": " machine learning")") Filters Applied: Conferences Journals 2010–2021"	434
ACM Journal	[[All: "software defect prediction"] OR [All: "software fault prediction"] OR [All: "software bug prediction"] OR [All: "software defect forecasting"]] AND [All: "artificial intelligence"] AND [All: "machine learning"] AND [Publication Date: (01/01/2010 TO 12/31/2021)]	90

5.1. *Various approaches for finding defects*

Once the development phase is over, every newly developed software product gets tested. The purpose of testing is to ensure bug-free delivery to the end-users. The more the code changes, testing efforts are higher. Based on the historical training dataset, we can classify the approaches into two broad categories at a high level, detecting the actual defect and predicting the possible defect with some confidence. Two approaches are:

1. Defect finding using Legacy approaches
2. Defect finding using Prediction approaches

Table 8 shows the high-level difference across various approaches.

We termed defect detection as the Legacy Approach. The literature outcome also discusses the evaluation parameters, which generally indicate the classifier's performance. In this SLR, we explore the various publicly available dataset. We try to identify the challenges inherent in the dataset and what kind of data validation methods have been used in earlier research? We have also analyzed the existing tools dedicated to Software Defect Prediction; their comparative analysis is presented. We have proposed a futuristic direction at the end of this literature outcome, considering the gaps identified in the earlier research.

Table 6
Criteria for inclusion and exclusion of research studies for Defect Prediction.

Criteria	Topic	Inclusion criteria	Exclusion criteria
1	Defect prediction dataset	Standard dataset, Or Custom dataset using open-source projects	Paper with no references to the dataset is filtered.
2	Artificial Intelligence, Machine Learning	Selected papers based on Machine Learning and Artificial Intelligence	Static analysis, Manual defect testing, and other unit testing automation are filtered.
3	Tools and framework-work used	The study employed conventional tools such as WEKA and others for feature selection and data modeling.	Paper was filtered, not mentioning the references for features used to build the classifiers.
4	Class Imbalance Treatment, Feature Selection, Usage of Software Metrics	Papers addressing Class Imbalance, Feature Selection, Dataset Validation, Software Metrics, Machine Learning, Deep Learning included.	Paper was filtered, which did not match the formulated research questions mentioned in Table 2.

Table 7
Search results from various databases.

Database	Search Result	Selected for review
ACM	251	67
Science Direct	360	23
Scopus	630	139
IEEE	434	48
Miscellaneous	8	6
Total	1681	283
146 out of 283 selected after removing duplicates from the above-selected papers across sources.		

Table 8
Comparison between traditional and predictive approaches.

Method	Approach	Process	Initial Cost	Error	Scope of Finding Defects	Extra Setup	Time	Historical Data Need
Manual Testing	Detection	Manual	Low	Possible Human Error	Limited	No	Take lots of time	No
Automation	Detection	Programmatic Execution	Medium	Coding errors possible	Scale well	Yes	Moderate time	No
Static analysis	Detection	Rule-based execution	Low	Mostly accurate	Limited	Yes	Less time	No
Peer Code review	Detection	Manual Process	Low	Mostly accurate	Limited	No	Take a lot of time	No
AI Approaches	Prediction	High computing process	High	Mostly Accurate, Depends on the data quality and techniques used	Wide possibility of Predictions	Yes	Moderate	Yes

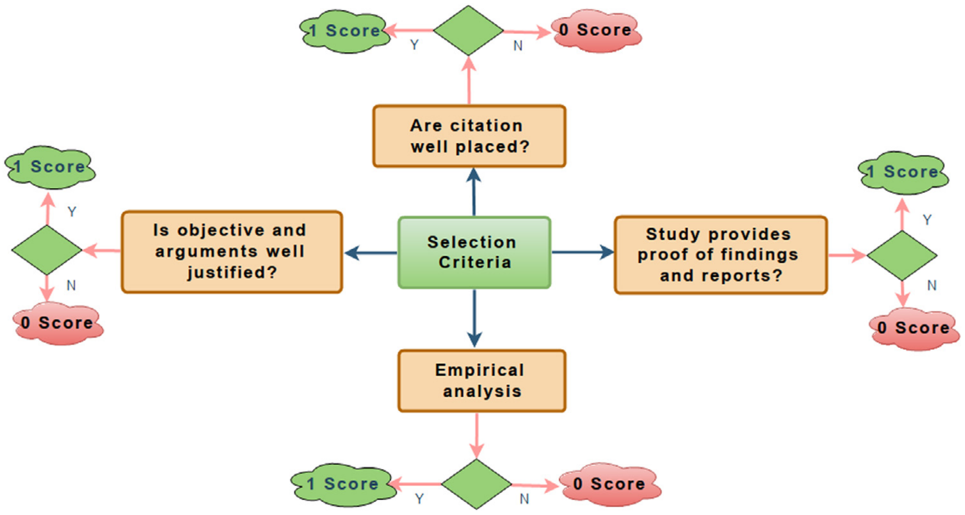
1 Score
Y
N
0 Score
Are citation well placed?
1 Score
Y
N
0 Score
Is objective and arguments well justified?
Selection Criteria
Study provides proof of findings and reports?
1 Score
Y
N
0 Score
Empirical analysis
1 Score
Y
N
0 Score

Fig. 10. Criteria for choosing a research article.

5.1.1. Defect detection using legacy approaches

These are the legacy approaches used for more than decades and are still quite popular in finding software defects. Detection approaches can be further classified: Manual test case execution (Jayanthi and Florence, 2019), automation for defect detection (Shen and Chen, 2020), running static code analyzers (Dong et al., 2018), manual code review, or peer review (Wahono et al., 2014). Fig. 12 shows various approaches for finding defects.

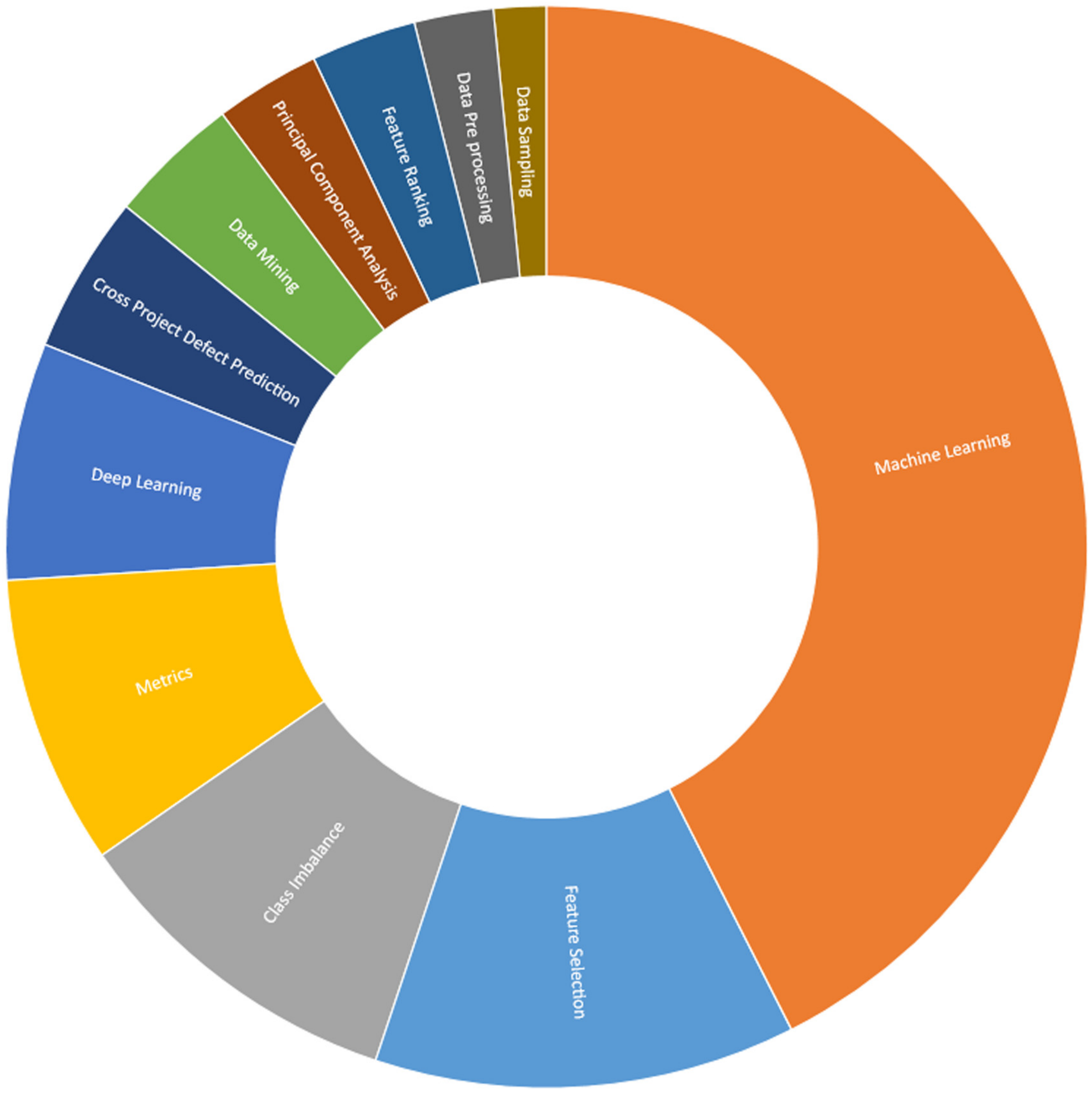

Fig. 11. Topic distribution across the selected studies for review.

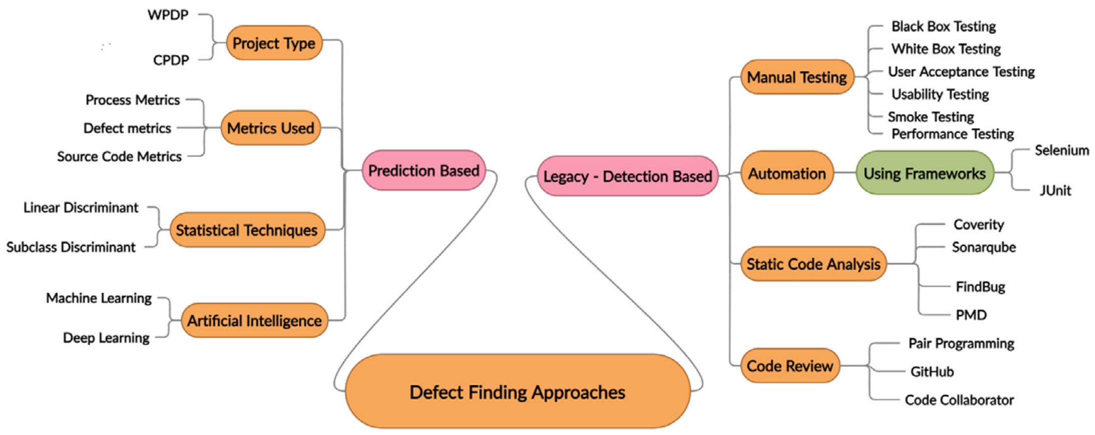

Fig. 12. Various methods for detecting defects.

1. *Manual testing* The software industry has adopted Manual testing (Jayanthi and Florence, 2019), where the Quality Assurance engineer executes business use cases designed based on the customer's business requirements and tries to figure out any defects in the software while executing those cases. The major bottleneck in this approach is it is a very time-consuming process, limiting how much testing can be done before releasing the product. As per the study (Jayanthi and Florence, 2019), manual testing required 27% of the total software development time. Particularly, in the product-based software companies, products are quite old and contain a huge code base with so

many product features that cannot be manually tested. Limited manual testing leads to hidden defects, which are not caught in manual testing. Manual testing can be further divided into various categories like:

Black-box testing

The tester does not know the code implementation in Black-Box testing, and his focus is to ensure that the desired functionality works as expected. Integration testing is an example of Black-Box testing.

White-box testing

In white-Box testing, the tester is aware of the inner workings of the code and has prior knowledge of how it is implemented. White-Box testing includes unit testing as an example.

Smoke testing

The purpose of the Smoke test is to check the sanity of the software product before moving it to production. It is the last test before putting the code in the production system.

User acceptance testing

User acceptance testing is done by the limited set of the actual user to check if they can use the software and see no issues.

Usability testing

Usability testing checks the ease of software usage, look and feel, and user interaction to ensure users have a smooth experience using the software product.

Performance testing

It is a kind of load test done to ensure the scalability of the software to the defined peak load, which is possible in the actual user environment.

2. *Using automation*

As manual testing needs Quality Assurance resources, it is costly and time-consuming. Automation testing technologies (Shen and Chen, 2020) are considered an important component for finding defects to reduce time and cost. Survey (Shen and Chen, 2020) referred to similar concepts for finding security defects using Automation techniques. Such tools execute the various test cases similar to manual execution but driven via software frameworks like Selenium. Those are fast in executions and give satisfactory results.

3. *Static code analysis for finding issues*

Static code analysis (Dong et al., 2018) analyzes the source code and checks against the predefined rules, mostly written in the XML documents. According to the rule, if the parser identifies a code pattern that matches the pre-defined rule, the code is marked as Buggy. Various tools like Check Style, PMD, FindBug, Sonarqube, and Coverity are examples of tools available for detecting the issues by statically analyzing the code for the software written in JAVA and C++ languages.

4.*Manual code review*

Manual code review is a manual process where one developer reviews the code written by another developer before committing the changes to the repository. Peer code review is a very commonly used technique for detecting issues in the code. Generally, code is getting reviewed by two other developers, and their approval is required before committing the changes.

5.1.2. *Defect finding using prediction approaches*

Defect prediction using Artificial Intelligence techniques has gained good focus, and many researchers have conducted their research for predicting software defects using Artificial Intelligence techniques. Defect prediction approaches can be classified as below:

(i) Classification based on the software project.
(ii) Classification based on the metrics used for defect prediction (Yu et al., 2020)
(iii) Statistical classification approach (Ma et al., 2014; Jing et al., 2016; Zhang et al., 2015; Gao et al., 2015b)
(iv) Classification based on the Artificial Intelligence techniques used for prediction

(i) *Choosing software project for prediction*

There are various forms of software development. It might be a new fresh development or an incremental release. Do we have previous versions of the software? We can divide the software project into four categories that are relevant for Software Defect Prediction:

1. Within Project Defect Prediction (WPDP)
2. Cross-Project Defect Prediction (CPDP)
3. Cross Version Defect Prediction (CVDP)
4. Heterogeneous Defect Prediction (HDP)

Within project defect prediction

WPDP has historical data available from the same project, so we can use the defect dataset of the earlier release to foresee defects in the new code. Using Artificial Intelligence techniques, we can predict if the code changes in a specific class will lead to a defect. The historical dataset could train the Machine Learning model to predict the defect if the modified class had any defect reported in an earlier release.

Cross-project defect prediction

When we do not have the previous release of the software for reference, we can take the defect prediction across projects. This approach uses the other completed project's historical dataset to predict the defect in the new project. Automated parameter optimization (Li et al., 2020) can significantly improve the prediction performance for cross-project defect prediction. Using Combined Approach CPDP + WPDP (Tabassum et al., 2020), where using Cross project data and then gradually adding the Within Project data, as and when code is available from the same project, improves the G-Mean by 53.90 percent.

Cross version defect prediction

In CVDP, earlier versions of the products are available and can be leveraged to predict the defects in the current or future version of the product. Study (Xu et al., 2018a) mentions using the Sparse Subset Selection to map the early version modules to represent the current version of the modules to predict the defects.

Heterogeneous defect prediction

Heterogeneous Defect Prediction uses multiple projects to train the classifier to predict defects in the new sample. HDP uses metrics from various projects and generally involves large data samples for training.

(ii) *Metrics classification*

One of the classifications of Defect Prediction is based on the selected metrics for training the classification model. It could be process-related metrics that depend on which process is followed for software development defect metrics or source code metrics. Software Metrics with strong discrimination ability can improve the classifier's efficiency. As a result, selecting the appropriate software metrics for Software Defect Prediction is critical. A few academics attempted to map defects directly to source code, utilizing existing source code metrics. They also incorporated their novel metrics to do static analysis for predicting the flaw in a fresh sample. On NASA, PROMISE, and Eclipse Repository datasets, the use of these metrics resulted in higher prediction in terms of Recall, F-Measure, and Precision. Response for class (RFC), lines of code (LOC), and lack of coding quality (LOCQ) are the most effective metrics, according to a study (Okutan and Yıldız, 2014) conducted on the PROMISE repository with nine open-source datasets.

In contrast, the number of children (NOC) and depth of inheritance tree (DIT) have very limited impact and are untrustworthy. Gray et al. (Gray et al., 2009) used static code metrics to investigate the classification's performance. Zang et al. (Zhang et al., 2010) employ static code metrics and training samples with incorrect class labels. The study discovered that while training a classifier, a dense defect

Table 9
Various metrics.

Process Metrics	Hosseini et al. (2017)	Loc	Punitha and Latha (2016), Ren et al. (2014), Chen et al. (2019c)	Product metrics	Hosseini et al. (2017)
Static code metrics	Bashir et al. (2018)	McCabe	Punitha and Latha (2016), Yousef (2015), Ren et al. (2014)	Object Oriented Metrics	Bashir et al. (2018), Rodriguez et al. (2013), Hosseini et al. (2017)
Micro Interaction Metrics (MIMs)	Taek et al. (2016)	Combined Metrics	Bashir et al. (2018)	Halstead	Punitha and Latha (2016), Yousef (2015), Ren et al. (2014)

dataset outperformed a normal dataset. D'Ambros et al. (D'Ambros et al., 2012) used source code metrics and change metrics to benchmark the performance of classifiers. Several software metrics in various combinations can be applied for classification, but no formal guidance was established. Because a training model that includes all metrics is computationally expensive, a guideline for selecting software metrics was needed. To pick the suitable software metrics, H. Bahadur et al. (Yadav and Yadav, 2015) created a membership function. To improve the accuracy of software failure prediction, Prasad et al. (Prasad et al., 2015) used software metrics with a variety of data mining approaches. Table 9 shows important metrics.

Process metrics usually define the attributes related to prediction and part of the process. For example, those could be Code changes, the number of developers, and the number of revisions in the same file for Software Defect Prediction. Some examples of Code Metrics are — McCabe, Halstead, and Object-oriented metrics (Yu et al., 2020). Source code metrics are usually based on the software development language with standard criteria to check the possibly defective code. For example, in JAVA, Object-Oriented source code metrics can help see the possible Buggy code. Some of the significant attributes for Object-Oriented metrics are Class associations or coupling, Depth of inheritance, number of classes, and number of public methods. Other than these standard metrics, a few more metrics are based on the complexity of the changes (D'Ambros et al., 2012), which try to measure the distribution of code changes. The higher distribution of the code changes indicates the higher possibility of the defect. Apart from the complexity, another attribute named Code Churn (D'Ambros et al., 2012) also indicates the quality or stability of the code. The higher the Code Churn, the more defects and need more testing and verification, whereas low code churn indicates the stable production-ready source code. Previous defect metrics can be captured from the defect dataset with information like the number of defects per line of code, defect density, severity, etc. For predicting defects, metrics related to defects are quite useful. Generally, defect metrics are used along with the other software metrics to boost the accuracy of the prediction.

(iii) *Statistical classification approach for defect prediction*

Discriminant analysis is a statistical approach that tries to classify the dataset into non-overlapping groups, which can be further used to classify the data into well-known categories. The Discriminant analysis is used in very few studies to separate the non-overlapping samples from a given dataset. Discrimination is usually performed based on the score of the quantitative predictor variable. Types of Discriminant analysis used in early research are Linear Discriminant Analysis and Subclass Discriminant Analysis.

(iv) *Artificial intelligence based approaches*

Artificial Intelligence techniques such as Machine Learning and Deep Learning in Software Defect Prediction have increased considerably in the last ten years. Fig. 13 shows major Artificial Intelligence techniques used for Software Defect Prediction.

Many researchers did their analysis based on Artificial Intelligence techniques with the different combinations of algorithms, datasets, features, and evaluation parameters to explore the usefulness of the Artificial Intelligence techniques for Software Defect Prediction. Early research has attempted to use many prominent algorithms for software defect classification. On a broad level, these algorithms can be categorized into three major types based on the learning style: Supervised, Unsupervised, Semi-Supervised. As per the survey, supervised learning algorithms are mostly used for building the classification model for Software Defect Prediction. Semi-Supervised learning is not used much in early research. Modified Co Forest (Punitha and Latha, 2016) solves the class imbalance issue, although it does not have a significant footprint. Unsupervised learning (Gong et al., 2019; Wenjie, 2019; Ali et al., 2020; Abdulshaheed et al., 2019a) is used less in previous research. The algorithms used are K-Means and KNN, which do not need labeled data for training and are very useful for the new projects where we do not have a historical dataset available for training. Artificial Intelligence-based learning algorithms can also be classified as Machine Learning and Deep Learning. Table 10 shows the comprehensive analysis of the various approaches.

a. *Machine learning*

One of the essential artificial intelligence approaches is Machine Learning. The training dataset is utilized in Machine Learning to build the classification model, which predicts future outcomes for the unknown samples. Some common examples are:

1. Linear Regression
2. Naive Bayes
3. Logistic Regression
4. Decision Tree
5. SVM
6. KNN
7. K-Means
8. Random Forest etc.

Various research utilizing Machine Learning and Deep Learning methodologies has been undertaken to measure the efficacy of classifiers and investigate the effects of data quality, feature selection, sampling strategy, and the usage of various Software Metrics. Exhaustive and heuristic search approaches for Software Defect Prediction were studied in Di Mario et al. (2021) for intrusion detection with excellent classification skills. Machine Learning approaches were used in Khamis and Gomaa (2015) to develop various score functions for drug design, while in Elmishali et al. (2018) Machine Learning was employed for automated planning and diagnosis in troubleshooting bugs (Elmishali et al., 2018). One of the most often used classifiers in Software Defect Prediction is the Naive Bayes classifier. The NB classifier outperforms more complex classification models despite its simplicity. Many researchers have utilized NB as a benchmark classification method to compare their proposals to the NB classifier. Transfer Naive Bayes (Ma et al., 2012) utilizes weighted NB mode for CPDP utilizing NASA datasets, resulting in a higher AUC and lower runtime cost. The prediction performance is improved when NB is combined with sampling (He et al., 2015). An association mining-based technique combined with the NB classifier (Rana et al., 2015) enhanced Recall on the PROMISE dataset. The K-Means (Wenjie, 2019) algorithm identifies negative class samples to determine their centroid, and it is combined with SMOTE to solve the class imbalance. The simple regression-based classification algorithm Logistic Regression (Hall et al., 2011; Xia et al., 2016; Zhang et al., 2018b; Cruz and Ochimizu, 2009) performs

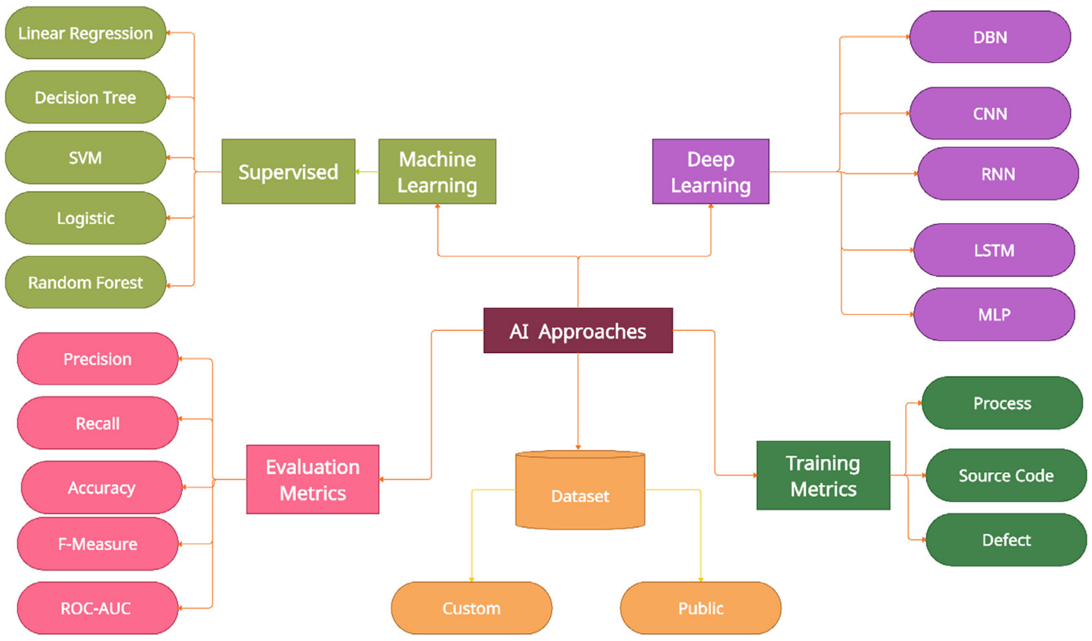

Fig. 13. Artificial Intelligence techniques for Software Defect Prediction.

better with Feature Selection. The use of LR techniques has been established in CPDP investigations. KNN (Okutan and Yildiz, 2016; Yu et al., 2017b; Zheng et al., 2020; Abdulshaheed et al., 2019a) improves the classifier's performance when used in conjunction with Feature selection. Precomputed Kernel produces superior performance. According to the study (Abdulshaheed et al., 2019a), KNN outperforms RF and MLP on the NASA dataset. For identifying software metrics, SVM (Peng et al., 2009; Catal, 2014; Huda et al., 2017; Okutan and Yildiz, 2016; Choeikiwong and Vateekul, 2016) is utilized in a semi-supervised learning approach in conjunction with ANN. According to a study (Peng et al., 2009), SVM is one of the top three ranked classifiers for predicting software defects. The notion of threshold adjustment is used in R-SVM (Choeikiwong and Vateekul, 2016) to mitigate a majority class bias. One of the best-performing ensemble approaches is Random Forest. It is not overly affected by optimization parameters. The Random Forest performs better than other classifiers in performance stability with an imbalanced dataset. Random Forest (Naseem et al., 2020; Tantithamthavorn et al., 2018; Wenjie, 2019; Laradji et al., 2015; Akour et al., 2017; Yu et al., 2017a) is an algorithm for ensemble learning. Random Forest improves when the class imbalance is corrected. Random Forest performs well for classification and regression. Random Forest performs better than other classifiers when the class imbalance is not addressed. According to the survey, most studies used Logistic Regression, Support Vector Machine, Decision Tree algorithm C4.5, C5.0, Naive Bayes, k-Means. These algorithms can be classified as the most widely used learning algorithms across various studies. Table 11 highlights the use of various Machine Learning approaches in early studies and the results of those studies.

b. *Deep learning*

Deep Learning is a more advanced technique that tries to imitate the working style of the human brain. Artificial neural networks are algorithms inspired by the structure and function of the brain, and deep learning is a branch of Machine Learning that deals with them. Data is analyzed with many layers, with every layer generating more simplified data, which the other layer can process, and the outcome can be produced more accurately. Some common examples are:

1. Convolutional Neural Networks (CNNs)
2. Recurrent Neural Networks (RNNs)
3. Multilayer Perceptron's (MLPs)
4. Deep Belief Networks (DBNs)
5. Long Short-Term Memory Networks (LSTMs) (Hoa et al., 2019).

Table 12 highlights the use of various Deep Learning approaches with the results of those studies.

The neural networks, particularly CNN and DNN, and their variants like RNN, were the most common deep learning algorithms employed in previous studies. Before implementing deep learning classification, most studies applied automatic feature extraction. As a result, Feature Selection is automatic with Deep Learning approach. AUC is the most used method of evaluation for Deep Learning. In a study (Hoang et al., 2019), CNN was used to forecast time for OpenStack projects and showed a 13.69% improvement over traditional Machine Learning algorithms. DNN was employed in a study (Dong et al., 2018) for Android applications, and it outperformed SVM and Naive Bayes. For decompiled Android apps, the AUC is .85. Attention-based Recurrent Neural Network was used in the study (Fan et al., 2019). It parses the code's AST to extract syntactical and semantic properties, which are then used in the ARNN-based second stage of classification. The data reveals a 14 percent increase in F-Measure and a 7% increase in AUC. Probabilistic Neural Network (PNN) (Pendharkar, 2010) was used with a hybrid approach to solve the classification problem for Software Defect Prediction. The most utilized deep learning algorithms across multiple studies are Perceptron, Multilayer Perceptron, Convolutional Neural Networks, and Recurrent Neural Networks. Table 13 shows a brief comparison of the few significant Non-Supervised Machine Learning approaches.

Table 14 shows a brief comparison of the significant Supervised Machine Learning approaches.

Table 15 shows a brief comparison of significant Deep Learning techniques.

Table 16 lists a few of the high-quality papers cited in this SLR that cover various areas of Software Defect Prediction.

c. *Evaluation parameters*

Most research has used evaluation parameters like Precision, Recall, F-Measure, and Probability of False Alarm to assess the quality of a classifier. Peng et al. (Peng et al., 2009) investigated and produced performance metrics for assessing the classifier's quality. AUC, PF, F-Measure, Recall, and Precision are the most commonly used CPDP evaluation criteria. Rathore and Kumar (Rathore and Kumar, 2019)

Table 10
Various defect identification and prediction approaches for finding defects.

Paper	Method	Approach	Advantages	Disadvantages
Jayanthi and Florence (2019)	Manual Testing	Legacy (Detection based)	1. No extra setup is needed 2. Quick feedback to the developer 3. No need for historical data	1. Human factor-error prone 2. Less number of defects 3. Costly 4. Time-consuming
Wahono et al. (2014)	Peer Code review	Legacy (Detection based)	1. It is mostly accurate, as reviews are done with the experts. 2. No need for historical data	1. Time-consuming 2. Find limited defects 3. Human factors.
Shen and Chen (2020)	Automation Testing	Legacy (Detection based)	1. It is fast than manual testing. 2. No human error, although the test code itself may have an error. 3. It can be linked with the code changes and the build process to quickly update developers. 4. No need for historical data	1. The initial cost is high 2. Need Extra setup 3. Find limited defects as per the automated test cases.
Dong et al. (2018)	Static code analysis	Legacy (Detection based)	1. It can be integrated into the developer's environment, quickly updating on defects. 2. No need for historical data	1. Based on predefined rules (code metrics) 2. Find only generic code defects 3. Cannot find the functional defects 4. Tied with the programming language
Felix and Lee (2017), Manivasagam and Gunasundari (2018), Zhang et al. (2018b), Gong et al. (2019), Wenjie (2019), Li et al. (2012), Laradji et al. (2015), Ali et al. (2020), Abdulshaheed et al. (2019a), Naseem et al. (2020), Choeikiwong and Vateekul (2016), Tantithamthavorn et al. (2018)	Machine Learning	Prediction based	1. Can predict many defects with higher confidence. 2. It can be useful to provide direction to utilize the resource in the area with the maximum likelihood of getting defects. 3. Easy to interpret	1. Defect prediction is possible to have a defect and not necessarily a defect. 2. Need a historical dataset 3. Need Extra setup 4. The initial cost is high 5. Feature selection is critical
Phan et al. (2018), Shen and Chen (2020), Liang et al. (2019), Gao et al. (2014), Ali et al. (2020), Abdulshaheed et al. (2019a), Fan et al. (2019), Qiu et al. (2019)	Deep Learning	Prediction based	1. Deep Learning can provide more accurate predictions for complex, big projects. 2. Less dependency on the feature selection.	1. Complex to understand the inner working of internal layers. 2. It is expensive computationally.
Ma et al. (2014), Jing et al. (2016), Zhang et al. (2015), Gao et al. (2015b)	Discriminant analysis (Statistical)	Prediction based	1. Less complex 2. Easy to implement 3. Works the best when data is separable.	1. Not effective when data is not separable. 2. Performance is not as good as other ML/DL techniques.

looked into the relationship between product metrics and fault proneness, which could be important information for improving forecast accuracy in the context of a project. Table 17 shows the evaluation parameters referred to in previous research.

5.2. Available datasets, are they good enough for multi-label predictions?

Earlier research publications are examined in this section to compare the use of publicly available datasets and the challenges faced for predicting software defects. It is also analyzed if these datasets can predict defect estimates, resource allocations, and the code to be fixed for the predicted defects. Most of the early researchers referred to NASA, PROMISE, and AEEEM datasets in their studies. KC3, PC4, CM1, PC3, KC1, and PC1 are the most used datasets in the early research. Apart from these, JIRA is a popular bug tracking system utilized as a bug repository by many open-source software. Yatish, Suraj, et al. (Li et al., 2020) created the bug dataset using the JIRA Bug repository. Fig. 14 shows the commonly used datasets from NASA, PROMISE, and AEEEM Repository.

Table 18 shows the various attributes of the dataset available publicly.

Table 19 shows the attributes of the JIRA dataset created by Yatish, Suraj, et al. (Yatish et al., 2019) for Software Defect Prediction. Table 21 lists the literature review references for the datasets.

5.2.1. Datasets challenges

Most of the datasets have the class imbalance issue, except a few datasets that are better in class imbalance, like JDT.Core, KC4, Eclipse 3.0, Xalan, whereas the most imbalanced datasets are CM1, JM1, PC5, LC, KC3, MC1, PC1, PC2, 90% or more datasets is skewed either towards defective or non-defective. Besides this, high dimensionality is another concern that impacts the learning algorithm's performance. Most of the datasets are good for binary classification but lack the adequate features for defect estimation, severity, resource allocations, code changes required to predict the defects. Fig. 15 shows the challenges in the available Software Defect Prediction datasets.

(i) *Class imbalance*

For Machine Learning classification, the training dataset should have an approximate equal positive and negative label, but that is seldom in any real-life business applications. Class Imbalance is a serious challenge that harms the prediction. The survey shows the attempts done in early research to develop a good Machine Learning classifier and mitigate the Class Imbalance. The techniques used for solving Class Imbalance are Bagging, Over Sampling, Under Sampling, Synthetic Minority Over-sampling Technique SMOTE, Two-Stage Cost-Sensitive Learning. The term "Class Imbalance" refers to a Machine Learning situation where the total number of positive data classes is significantly less than the total number of negative data classes.

Table 11
Machine Learning approach used in earlier research.

Paper	Year	Learning	Dataset	Evaluation Parameters	Algorithm	Result with the best performance
Yu et al. (2020)	2020	ML	Open Source Java projects.	AUC	KNN, LR, NB	LR performs the best with PPM with AUC 0.88 for the Jedit project.
Wu et al. (2020)	2020	ML	PROMISE NASA	RECALL, AUC	AdaBoost, DecisionTree	AdaBoost and Decision Tree performs better for the synapse-1.2 dataset with recall value as – 0.55 and 0.554.
Naseem et al. (2020)	2020	ML	NASA	F-MEASURE, PRECISION, RECALL	NB, RF, KNN, MLP SVM, J48	RF generates better results with 0.99 on the PC2 dataset.
Ali et al. (2020)	2020	ML	NASA	F-MEASURE ACCURACY MCC	Ensemble learning, NB, MLP, RBF, SVM, KNN, DT, RF	Ensemble approach outperform with F-Measure = 0.75 Accuracy = .87, MCC = 0.669.
Saifan and Abu-wardih (2020)	2020	ML	PROMISE	AUC	KNN (Bagging based)	KNN with Bagging with Feature selection using PCA performs the best with AUC=0.726 for the CM1 dataset.
Abdulshaheed et al. (2019a)	2019	ML	NASA	MAE, RMSE, RAE, RRSE, R Square	KNN RF MLP	The result indicated the KNN performs better with R square value = 0.9969.
Wenjie (2019)	2019	ML	UCI Machine Learning	F-Measure, G- Mean	K-Means With SMOTE	KMS-SMOTE has a better F- Measure of 0.9127 with the Ionosphere dataset.
Ji et al. (2019)	2019	ML	PROMISE	F-Measure	Weighted Naive Bayes (WNB-ID)	Results are better with WNB-ID with F-Measure = 0.8669 for the POI-2.5 dataset.
Chen et al. (2019a)	2019	ML	ALEEM NASA ECLIPSE	Wilcoxon sign-rank, AUC	J48, Random Forest	With Multiple view transfer learning, a *p*-value of RF =0.8964 for the eclipse dataset.
Tantithamthavorn et al. (2018)	2019	ML	NASA PROMISE	Precision, Recall, F- Measure AUC	C5.0, xGBTree, and GBM)	For Optimized xGBTree, the average rank is 2.39, whereas for C5.0, it is 2.56, and for GBM, it is 2.94.
Ji et al. (2018)	2018	ML	NASA	Precision, Recall, F- Measure, AUC	Two Stage Naive Bayes (TSWNB)	TSWNB performs better with AUC value=0.7835 for the PC4 dataset.
Xu et al. (2018b)	2018	ML	NASA AEEEM	F-Measure AUC, Balance	LR, NB, KNN, RF	Naive Bayes with Kernel distribution achieves better results. F- Measure =0.907 for WPDP.
Ghosh et al. (2018)	2018	ML	Open-Source Repository	AUC, Accuracy, F-measure, Percentage	BBN NB J48	For nonlinear MDTs, model performance is better. The accuracy of the J48 for the Apache dataset is 74.2268 percent.
Zhang et al. (2018b)	2018	ML	PROMISE	F-Measure	Max, Boosting J48, RF, LR Boosting Naive CODEP (Logistic)	RF, BoostingJ48, and Max have better F-scores than the F-measure of the CODEP(Logistic) by 2.33%, 0.33%, and 36.88%, respectively.
Bowes et al. (2018)	2018	ML	NASA	Precision, Recall, F- Measure	RF, NB, RPart, SVM	Classifier ensembles with decision-making strategies perform best in defect prediction.

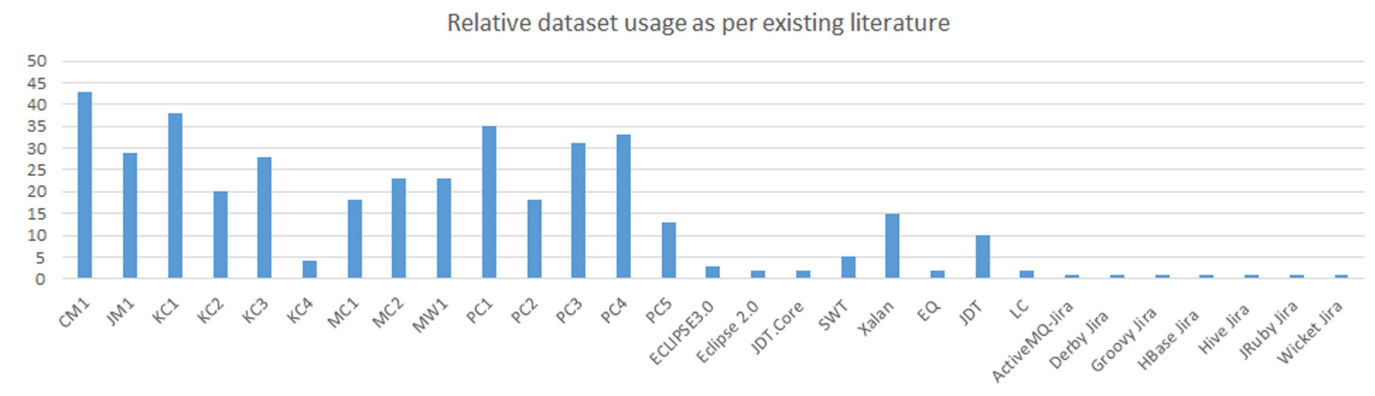

Fig. 14. Commonly used Datasets from NASA, PROMISE, AEEEM repositories.

Fig. 16 shows the percentage of defective instances in various datasets from NASA, PROMISE, AEEEM, and JIRA repository.

The Class Imbalance is addressed in a variety of ways by researchers. Resampling approaches that combined Threshold Moving

Table 12
Deep Learning approach used in earlier research.

Paper	Year	Learning	Dataset	Evaluation Parameters	Algorithm	Result with the best performance
Qiu et al. (2019)	2019	DL	PROMISE	Precision Recall F-Measure	TCNN DBN	TCNN performs better. F-Measure for Log4j dataset is 0.702.
Liang et al. (2019)	2019	DL	PROMISE GitHub	Recall, Precision, F-Measure	LSTM Network	The proposed LSTM approach improves DBN ISDA approaches by 8.2
Fan et al. (2019)	2019	DL	Open Source Apache	F-Measure AUC	DP-ARNN	F1 measure is increased by 14% using DP-ARNN, whereas a 7% increase is observed in the AUC compared to the other standard methods.
Gao et al. (2014)	2014	DL	PROMISE Eclipse	AUC	MLP	MLP performs better with boosting on all the studied datasets.
Dong et al. (2018)	2018	DL	Open Source GitHub	AUC	DNN	85.98
Gao and Khoshgoftaar (2015)	2015	DL	Eclipses	Precision, Recall, F-Measure, AUC	MLP with Feature Selection Sampling	MLP performs the best with an AUC value of 0.8501 for the SMO35 sampling technique.

Table 13
A brief comparison of unsupervised Machine Learning algorithms.

Papers	Algorithms	Advantage	Disadvantage
Gong et al. (2019), Wenjie (2019)	K-Means	1. Easy implementation 2. Suitable for large datasets.	1. Do not perform well with high Dimensionality.
Ali et al. (2020), Abdulshaheed et al. (2019a)	K-NN (k-Nearest Neighbors)	1. Well suited for Multiclass classification. 2. Simple and easy implementation.	1. Slow when the dataset grows. 2. Better performance with fewer features.

Table 14
A brief comparison of supervised Machine Learning algorithms.

Papers	Algorithms	Advantage	Disadvantage
Felix and Lee (2017), Manivasagam and Gunasundari (2018)	Linear Regression	1. Simple and Effective 2. Tuning hyperparameters is not required	1. Poor results for non-linear data 2. Comparative performance is not as good as other algorithms
Zhang et al. (2018b)	Logistic Regression	1. Simple and Effective 2. Tuning hyperparameters is not required	1. Poor results for non-linear data 2. Comparative performance is not as good as other algorithms
Naseem et al. (2020), Li et al. (2012)	Decision Tree	1. No impact of missing values 2. Automatic Feature Selection	1. Over fitting issues 2. Data sensitivity 3. Take more time
Laradji et al. (2015), Choeikiwong and Vateekul (2016)	Support Vector Machine	1. No major impact of high dimensionality. 2. Best performance, when classes are separable	1. Slow if the dataset is large 2. Hyper parameter tuning required 3. A poor result in case of class overlap
Laradji et al. (2015), Tantithamthavorn et al. (2018)	Random Forest	1. Works well with an imbalance dataset 2. No over fitting 3. Well handling of missing values	1. The feature should be capable of prediction 2. Choosing the right parameter is important for good results

Table 15
A brief comparison of Deep Learning approaches.

Papers	Algorithms	Advantage	Disadvantage
Phan et al. (2018), Shen and Chen (2020), Qiu et al. (2019)	Convolutional Neural Network (CNN)	1. Efficient computation. 2. Auto Feature selection.	1. Need lots of training data.
Shen and Chen (2020), Fan et al. (2019)	RNN	1. Good for time series prediction	1. It is complex. 2. Training is difficult.
Liang et al. (2019), Cui et al. (2019), Fan et al. (2019), Hoa et al. (2019)	LSTM	1. Can predict time series with a time lag of unknown duration.	1. Require more time to train. 2. It needs more memory.
Gao et al. (2014), Ali et al. (2020), Abdulshaheed et al. (2019a)	MLP	1. Works well with noisy data. 2. It can also work with non-linear data.	1. It needs many parameters as it is fully connected.

Table 16
High-ranked papers for specific aspects of Software Defect Prediction.

Topic	Paper Title	Name of journal	Cited By	Cite Score	Algorithm/Techniques	Dataset
Class Imbalance	An Improved SDA Based Defect Prediction Framework for Both Within-Project and Cross-Project Class-Imbalance Problems (Jing et al., 2016)	*IEEE Transactions on Software Engineering*	100	14.9	Subclass discriminant analysis	PROMISE NASA ALEEM
Selecting a sample	Sample-based Software Defect Prediction with active and semi-supervised learning (Li et al., 2012)	*Automated Software Engineering*	127	6.2	Active and semi-supervised learning ACoForest	PROMISE
Feature Ranking	A feature selection approach based on a similarity measure for Software Defect Prediction (Yu et al., 2017b)	*Frontiers of Information Technology and Electronic Engineering*	15	4.3	KNN	NASA
Feature Selection	Software Defect Prediction using ensemble learning on selected features (Laradji et al., 2015)	*Information and Software Technology*	196	8.6	Ensemble learning algorithm	NASA
Data Pre Processing	Is "better data" better than "better data miners"?: On the benefits of tuning SMOTE for defect prediction (Agrawal and Menzies, 2018)	*International Conference on Software Engineering*	75	4.6	SMOTE and SMOTUNED	SEACRAFT
Deep Learning - LSTM	Seml: A Semantic LSTM Model for Software Defect Prediction (Liang et al., 2019)	IEEE Access	18	4.8	LSTM	GitHub
Artificial Neural Networks	Transfer learning using computational intelligence: A survey (Lu et al., 2015)	*Knowledge-Based Systems*	403	11.3	Neural network-based transfer learning	
Convolutional neural network	DGCNN: A convolutional neural network over large-scale labeled graphs (Phan et al., 2018)	*Neural Networks*	21	10.9	CNN	GitHub
Unsupervised Learning	Software defect number prediction: Unsupervised vs. supervised methods (Chen et al., 2019c)	*Information and Software Technology*	44	8.6	LOC metric	Open-source projects
Logistic Regression	A systematic literature review on fault prediction performance in software engineering (Hall et al., 2011)	*IEEE Transactions on Software* Engineering	642	14.9	Naive Bayes	NASA PROMISE
Principal Component Analysis	Software Defect Prediction based on kernel PCA and weighted extreme learning machine (Xu et al., 2019)	*Information and Software Technology*	50	8.6	Kernel Principal Component Analysis (KPCA) and Weighted Extreme Learning Machine (WELM)	NASA
Metrics	An empirical study on Software Defect Prediction with a simplified metric set (He et al., 2015)	*Information and Software Technology*	127	8.6	One-way ANOVA, Naive Bayes	PROMISE
Cross Project Defect Prediction	HYDRA: A massively compositional model for cross-project defect prediction (Xia et al., 2016)	*IEEE Transactions on Software Engineering*	162	14.9	Genetic algorithm Ensemble learning	PROMISE
Support Vector Machine	Empirical evaluation of classifiers for software risk management (Peng et al., 2009)	*International Journal of Information Technology and Decision Making*	65	4.0	SVM	NASA
Ensemble Learning	Multiple kernel ensemble learning for Software Defect Prediction (Wang et al., 2016)	*Automated Software Engineering*	69	6.2	Multiple kernel ensemble learning (MKEL)	NASA MDP

(Wang and Yao, 2013) with a dynamic variant of the AdaBoost Ensembling algorithm known as AdaBoost-NC outperformed regular AdaBoost. Experiments also showed that Random Forest and Naive Bayes performed better. The KPWE framework (Xu et al., 2019), which combines the KPCA and WELM, outperformed traditional algorithms in terms of F-Measure, MCC, and AUC for NASA and PROMISE datasets. Normalizing the data distribution is commonly used to tackle Class Imbalance, but another feature called Discriminant ability (Zhang et al., 2015) can also be employed to fix the problem. It employs EDBC's Dissimilarity-based classification. EDBC enhanced classification performance for Naive Bayes by 5 to 9.09 percent, Random Forest by 1.2 to 6.33 percent, IB1 by 9.21 to 18.57 percent, Multilayer Perceptron by 4.88 to 10.26 percent, and Logistic Regression by 8.86 to 13.16 percent. Random Under-Sampling, Random Over-Sampling, SMOTE, Bagging, and Boosting have been demonstrated to be less successful than EDBC. Model ASRA (Zhou et al., 2018b), which employs attribute selection, sampling, and ensemble approaches at various stages, produced a higher F-Measure for the UCI dataset when compared to a set of classifiers that did not use the ASRA model. Class imbalance is also decreased

Table 17
Various evaluation parameters for Software Defect Prediction quality.

Precision	The ratio of correctly categorized positive samples to the total number of samples categorized positively, including both correctly and mistakenly classified positive samples, is known as precision.	$\frac{True\ Positive}{True\ Positive + False\ Positive}$
Recall	The recall is calculated as the proportion of correctly categorized positive samples to the sum of correctly categorized true and false-negative samples.	$\frac{True\ Positive}{True\ Positive + False\ Negative}$
Accuracy	Accuracy is the ratio between the sum of True classified Positive and Negative versus the total number of Samples	$\frac{True\ Positive + True\ Negative}{Total\ Samples}$
F-Measure	F-Measure, a Harmonic Mean of Precision and Recall, is used to define the accuracy of Machine Learning categorization.	$2 \times \sqrt{\frac{Precision \times\ Recall}{Precision + Recall}}$
G-Mean	G-Mean (Geometric Mean) is a metric for comparing the categorization balance of majority and minority datasets. Low G-Mean denotes the weaker classification performance for the positive cases, although the negative cases are classified correctly.	$\sqrt{Sensitivity \times Specificity}$
AUC	The Area Under the Curve can measure the classification model's performance, distinguishing between Positive and Negative classes. The performance of the classification algorithm will be higher if the AUC value is higher.	
Probability of False Alarm (pf)	False Alarm in Software Defect Prediction means that the classification output predicts that the code is buggy even when not. As a result, the Probability of False Alarm can be used to assess the quality of a defect prediction model.	$\frac{False\ positive}{True\ Negative + False\ Positive}$

Table 18
Various attributes of the publicly available dataset.

Dataset	Source	Language	LOC	Attributes	Number of rows	Defective rows	Defective (%)	Imbalance ratio
CM1	NASA	C	17K	22	498	49	9.84%	10
JM1	NASA	C	457K	22	10885	8779	80.65%	1
KC1	NASA	C++	43K	22	2109	326	15.46%	6
KC2	NASA	C++	19K	22	522	105	20.11%	5
KC3	NASA	Java	8K	40	458	43	9.39%	11
KC4	NASA	Perl	25K	40	125	61	48.80%	2
MC1	NASA	C++	66K	39	9466	68	0.72%	139
MC2	NASA	C++	6K	40	161	52	32.30%	3
MW1	NASA	C	8K	40	403	61	15.14%	7
PC1	NASA	C	26K	40	1107	76	6.87%	15
PC2	NASA	C	25K	40	5589	23	0.41%	243
PC3	NASA	C	36K	40	1563	160	10.24%	10
PC4	NASA	C	30K	40	1458	178	12.21%	8
PC5	NASA	C++	162K	39	17186	516	3.00%	33
Eclipse 3.0	PROMISE	Java	1306K	198	661	415	62.78%	2
Eclipse 2.0	PROMISE	Java	797K	198	6729	2611	38.80%	3
JDT.Core	PROMISE	Java	181K	198	939	502	53.46%	2
SWT	PROMISE	Java	194K	198	843	208	24.67%	4
Xalan	PROMISE	Java	57K	20	886	411	46.39%	2
EQ	AEEEM	Java	70.4K	62	324	129	39.81%	3
JDT	AEEEM	Java	239.4K	62	997	206	20.66%	5
LC	AEEEM	Java	149.1K	62	691	64	9.26%	11
ML	AEEEM	Java	381.6K	62	1862	245	13.16%	8
PDE	AEEEM	Java	345.6K	62	1497	209	13.96%	7

Table 19
JIRA dataset created by Yatish, Suraj, et al. (Yatish et al., 2019) for defect prediction.

Dataset	Source	Language	LOC	Attributes	Number of rows	Defective rate%
ActiveMQ	JIRA	JAVA	142-299K	66	1884	6%–15%
Derby	JIRA	JAVA	412-533K	66	2705	14%–33%
Groovy	JIRA	JAVA	74-90K	66	821	3%–8%
HBase	JIRA	JAVA	246-534K	66	1059	20%–26%
Hive	JIRA	JAVA	287-563K	66	1416	8%–19%
JRuby	JIRA	JAVA	105-238K	66	731	5%–18%
Wicket	JIRA	JAVA	109-165K	66	1763	4%–7%

by employing the Decision Forest approach for cost-sensitive (Siers and Islam, 2018) prediction. The Cost-Sensitive Framework results reveal that Decision Forest has the lowest average cost for the NASA dataset. Omni-Ensemble Learning (OEL) (Mousavi et al., 2018) is an Ensemble Learning (OEL) approach that leverages the Over-Bagging approach to improve classification performance for an imbalanced dataset. Experiments demonstrate that applying OEL improves G-Mean, Balance, and AUC metrics. For OEL, the PF Measure yielded poor results. The EMR SD algorithm (He et al., 2019), a RIPPER-based Ensemble MultiBoost algorithm, employs a combination of Feature Selection and Sampling techniques. It aids in the decrease of variance. Results on the NASA MDP dataset reveal that EMR SD outperforms DNC and CEL in terms of Accuracy, F-Measure, AUC, and Balance. KMFOS (Gong et al., 2019) is a clustering-based oversampling with noise filtering strategy that uses

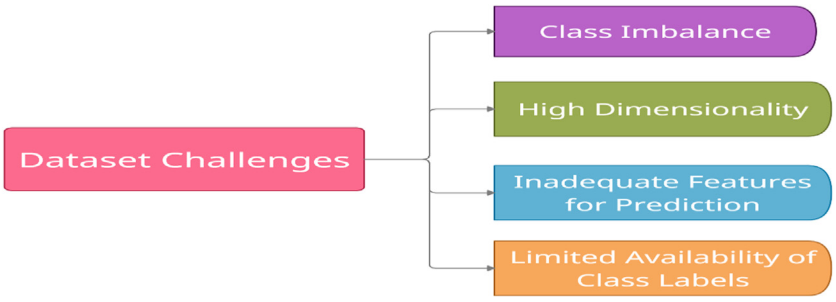
Dataset Challenges
Class Imbalance
High Dimensionality
Inadequate Features for Prediction
Limited Availability of Class Labels

Fig. 15. Dataset challenges for Software Defect Prediction.

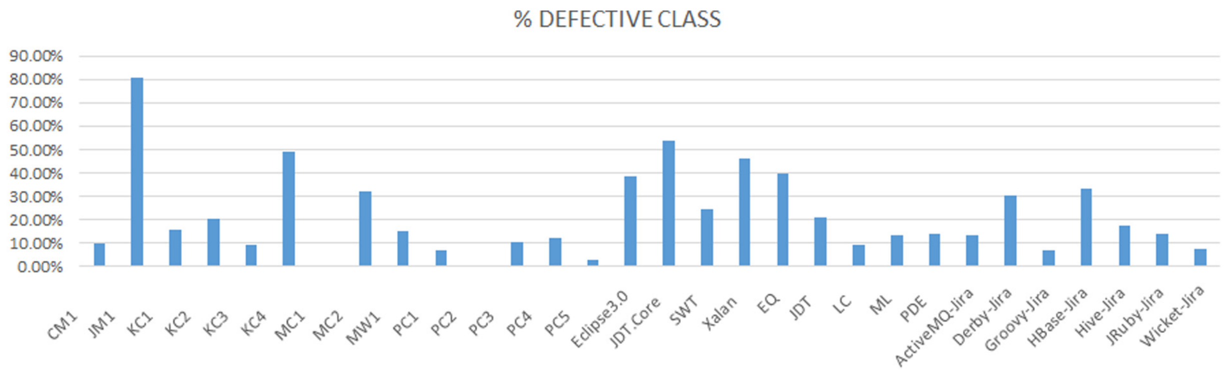

Fig. 16. Class imbalance: % defective class in the NASA, PROMISE, AEEEM, and JIRA databases.

the K-means algorithm to improve Imbalanced dataset classification performance. When KMFOS is compared with approaches such as Over Sampling, SMOTE, Balance Bagging classifier, Instance Hardness Threshold, and Cost Sensitive methods, experimental findings are better in Recall and Balance. Tree family algorithms (Naseem et al., 2020) such as the Hoeffding Tree (HT), Random Tree (RT), Credal Decision Tree (CDT), Cost-Sensitive Decision Forest (CS-Forest), Forest by Penalizing Attributes (Forest-PA), Logistic Model Tree (LMT), Random Forest (RF), Decision Stump (DS), REP-Tree (REP-T), and Decision Tree (J48) have also been found to be useful in resolving the Class Imbalance issue in the Software Defect Prediction dataset. Sun et al. (2012) focused on a new strategy for turning an unbalanced dataset into a multi-class dataset and then performing software fault prediction using the ensemble learning method, which produced improved results. Wang and Yao (2013) investigated numerous approaches for learning class imbalance and compared the results of the various methods. Few techniques to handle Class Imbalance are:

(ii) *Bagging*

Bagging (Wahono et al., 2014; Saifan and Abu-wardih, 2020), and (Punitha and Latha, 2016) is a kind of ensemble algorithm that uses various subsets of the training dataset and tries to fit multiple models, then combines the prediction from all the created models.

(iii) *Oversampling*

The characteristics and sample methods also influence the quality of the prediction. Data sampling helps resolve Class Imbalance concerns and has been demonstrated to improve prediction efficiency. Oversampling uses the minority classes and duplicates the records to balance the Defective versus Non-Defective rows from the dataset. Oversampling is usually combined with other techniques to avoid the over-fitting of the model. When training data are scarce, Random Over Sampling (Gao and Khoshgoftaar, 2015; Chen et al., 2019b) method is appropriate.

(iv) *Under-sampling*

Under-sampling is the exact opposite of over-sampling. To address the issue of class imbalance, rows from the majority classes are removed to equalize the majority and minority classes with in-training samples. Sampling techniques (Huda et al., 2018; Gao et al., 2015a; Khuat and Le, 2019; Wang and Yao, 2013) were adapted to reduce the number of samples that are not significant and sort of noise. Such reduction in the non-significant data also helps to reduce the class imbalance issue that otherwise negatively impacts the classification. Random Under Sampling (RUS) (Khoshgoftaar and Gao, 2009; Gao et al., 2014, 2015a; Gao and Khoshgoftaar, 2015; Khuat and Le, 2019; Gao et al., 2015b) is the most commonly used sampling method for Software Defect Prediction, and according to a study (Gao et al., 2014), it outperforms SMOTE. It is primarily implied to bring the class balance into equilibrium by lowering the majority.

(v) *SMOTE*

SMOTE (Bashir et al., 2018; Gao and Khoshgoftaar, 2015; Khoshgoftaar et al., 2015; Wenjie, 2019; Gong et al., 2019; Gao et al., 2015b; Choeikiwong and Vateekul, 2016; Chen et al., 2019b) is a more sophisticated technique based on random minority over sampling and further customized by combining with the K-Means algorithm to solve the marginalization problem (Wenjie, 2019). SMOTE produced superior results when combined with under-sampling. It is highly advised to use Feature Selection in conjunction with oversampling or SMOTE.

(vi) *Two Stage Cost Sensitive Learning*

A Two-Stage Cost-Sensitive Learning (TSCS) method, according to a study (Liu et al., 2014b), helps the resolution of Class Imbalance and High-Dimensional Data issues. Not only is cost information employed in the categorization stage, but it is also used in the feature selection stage. Three cost-sensitive feature selection methods, CSVS, CSLS, and CSCS, are developed by including cost information into standard feature selection algorithms. According to experimental data, the proposed TSCS approaches outperform single-stage cost-sensitive learning methods, whereas the proposed cost-sensitive feature selection methods exceed typical cost-sensitive Feature Selection methods.

(vii) *High dimensionality*

High dimensionality in the dataset indicates that the number of dimensions or features is quite high, making an inefficient calculation. The performance of the Machine Learning algorithm reduces due to high dimensionality. It is important to consider dimensionality reduction as part of data pre-processing before applying Machine Learning algorithms. The majority of the research focused on selecting the appropriate features and avoiding the insignificant features using the feature selection techniques.

(viii) *Inadequate features for prediction*

Predicting actionable items for the software development team is more valuable than simply addressing the binary classification problem of determining whether a piece of code, often a class, is defective. According to the survey dataset used in the early research, it is critical to examine if the training dataset has enough features that can be used in Multi-Label predictions such as estimations or story points, severity, resource allocation, possible remedy, and so on. The majority of the public datasets utilized in the early studies are only capable of binary classification and cannot make Multi-Label predictions.

(ix) *Inadequate class labeling*

According to the survey, most previous research focused on binary classification, which means predicting whether code is buggy or not. As a result, early studies did not focus on the significant features needed to predict additional information about the software defect. Although we can meet this requirement by selecting the relevant feature from the open-source dataset, there will still be another challenge due to the lack of labeled classes. Labeled classes are required to train the learning algorithm to predict the additional defect-related information, specifically for Multi-Label prediction. Existing studies are lacking in adequate class labels for Multi-Label prediction.

5.2.2. Feature selection/reduction

The essential characteristics in the training dataset capable of good classification are determined through Feature Selection. Using the proper features can improve the classification model's efficiency. It also reduces the amount of time it takes to generate the results. Feature selection is a technique for picking the most relevant feature while disregarding those not. Features in a software dataset must be carefully chosen to successfully identify problematic components. Feature selection excludes features that do not help categorize data and cause poor performance. For feature selection, early studies used a variety of strategies, like as-

1. Correlation-based Feature Selection (CFS) (Bashir et al., 2018; Zhang et al., 2015; Saifan and Abu-wardih, 2020; Laradji et al., 2015; Ali et al., 2020; Jakhar and Rajnish, 2018)
2. Correlation-based filter solution (FCBFS) (Zhang et al., 2015)
3. Clustering-based feature subset selection FAST algorithm (Zhang et al., 2015)
4. Forward Selection (Laradji et al., 2015)
5. Greedy Forward Selection (Laradji et al., 2015; Saifan and Abu-wardih, 2020)
6. Genetic Ant Colony Optimization (GACO) (Punitha and Latha, 2016)
7. Similarity Measures (Yu et al., 2017b)
8. Dynamic model based on Nonlinear Manifold Detection Techniques (Nonlinear MDTs) (Ghosh et al., 2018)
9. Chi-square (Bashir et al., 2018)
10. Information Gain Bashir et al. (2018), Cabral et al. (2019), Saifan and Abu-wardih (2020), Catolino et al. (2019)
11. Principal components analysis (PCA) (Bashir et al., 2018; Jayanthi and Florence, 2019; Saifan and Abu-wardih, 2020; Jakhar and Rajnish, 2018)
12. Kernel Principal Component Analysis (KPCA) (Xu et al., 2019)

AUC, Precision, Recall, F-Measure, Accuracy, PF, Sensitivity, and MCC are used to evaluate the performance of various classifiers with and without these Feature Selection strategies. In the research for Feature Selection studies, NASA and PROMISE databases are mostly explored. FAST (Zhang et al., 2015) outperforms other feature selection strategies, while correlation-based algorithms show only a modest improvement. Study (Agrawal and Menzies, 2018) concluded that it is more important to do the data pre-processing than choosing the classifier. Earlier research has seen a significant improvement in prediction accuracy by adopting Feature Selection techniques. According to a study (Laradji et al., 2015), the Greedy Forward selection performed best for NASA dataset PC2, PC4, MC1 with an AUC of around 1. GACO was recommended for the NASA MDP dataset in a study (Punitha and Latha, 2016), with greater Precision, Recall, and F-Measure than other approaches. In a study (Yu et al., 2017b), Feature Weights and Feature Ranking were utilized to choose features, and the KNN classifier produced a superior AUC for prediction. The accuracy of a dynamic model, based on Nonlinear Manifold Detection Techniques (Nonlinear MDTs) (Ghosh et al., 2018) proved higher, and the F-Measure using J48 was statistically significant. When paired with data balance and noise filtering, the Chi-Square test, Information Gain, and ReliefF (Bashir et al., 2018) yield better results. Principal Component Analysis (Jayanthi and Florence, 2019) revealed a higher AUC of 97.2 percent, while KPCA (Xu et al., 2019) revealed a higher MCC and F-Measure. On the NASA dataset, T. Khoshgoftaar et al. (Khoshgoftaar et al., 2010) investigated Feature Selection and Sampling. Wang et al. (Wang et al., 2010) investigated Ensemble Feature selection and analyzed Ensemble Feature selection effectiveness. The study compared approaches that select a single feature vs. methods that select multiple features. Khoshgoftaar et al. (2015) investigated several Feature Selection procedures and showed how they affect the prediction model. To manage a highly imbalanced Software Defect Prediction dataset, Gao and Khoshgoftaar (2015) coupled sampling strategies with Feature Selection methods. They evaluated several Sampling methods in combination with Feature Selection methods in their research work. Yu et al. used Feature Selection and Feature Ranking with Feature weight (Yu et al., 2017b) to eliminate the redundant features and enhance the prediction's efficiency and quality. Saifan et al. (Zheng et al., 2020) used ensemble methods to investigate class Imbalance and Feature Selection and assessed several Feature Selection techniques to improve the prediction model's performance. According to a study (Chen et al., 2020), applying Software Visualization, Deep Learning can eliminate the necessity for Feature extraction for Software Defect Prediction. Feature Reduction (Tiwari et al., 2017) significantly improved classification accuracy and processing time by reducing dimensionality. Dimensionality reduction can also be handled using Neural Networks (Lu et al., 2015). Principal Component Analysis (He et al., 2019; Zhang et al., 2018a) can help extract the relevant feature and drop the insignificant ones. Chi-square (Jakhar and Rajnish, 2018) is a statistical test, which can be used to see the association of the categorical value among the given dataset and can help select the appropriate features for the construction model. Principal Component Analysis (Xu et al., 2019; Jayanthi and Florence, 2019; Ren et al., 2014; Saifan and Abu-wardih, 2020; Ji and Huang, 2018) is a technique for reducing the dimensionality of a dataset to improve classification performance. Information Gain Saifan and Abu-wardih (2020), Siers and Islam (2015), Gao et al. (2014), Ji et al. (2019) helps decrease the uncertainty of the result and enhances the accuracy of the prediction by selecting the features that offer better gain than others. Gain is evaluated for each variable, and those variables are picked, which maximizes the information gain and improves the quality of the classification algorithm. The Correlation Coefficient can find the worth of the features using Pearson's Correlation (Saifan and Abu-wardih, 2020; Yu et al., 2017b; Laradji et al., 2015). It helps identify an optimized set of features that can train a classification model. It is difficult to determine which feature selection technique is the best because several researchers have found that a combination of different classifiers and datasets performs differently. However, the most commonly used Feature Selection strategies are Correlation-based and Principal Component Analysis. Good quality data is significant for getting a high-quality prediction. It is important to have good quality data, balanced and noise-free. There is no doubt that data pre-processing is a significant step in Machine Learning or Artificial Intelligence-related tasks.

5.3. Data validation techniques for software defect prediction modeling?

Dataset validation is critical since it assures that the data used for the prediction can accurately forecast the outcome. This section explores the data validation methods used in the earlier research. Artificial Intelligence techniques especially Supervised Learning, need labeled training datasets to train the classification model. The quality of the labeled dataset significantly impacts the classification model's performance. If we have incorrect labels or missing labels, the predicted result may be inaccurate. K Fold Cross-Validation is used for validating the appropriateness of the dataset samples.

5.3.1. K fold cross-validation

K Fold cross-validation is a technique for dividing datasets into K samples at random. From the K subsamples, one of the samples is used for validating the results, and the rest, K-1 samples, are used for building the classification model or used as a training dataset. K Fold Cross-Validation is usually used with values K=5 or K=10. The K-fold Cross-Validation technique can achieve the best prediction results by optimizing the selection of samples from the available dataset for training and validation. Apart from the other issues, having a generic sanity check on the dataset is recommended to detect the redundant rows, missing values before using the dataset or selecting samples for the model training. The survey outcome suggests that few research papers conducted data quality checks before classification steps. Although dataset validation is a significant task, it is lacking in early research. As an essential step in predicting software defects, it is advised to incorporate dataset validation to validate its applicability. It is suggested that increased emphasis be placed on data quality. If required, the necessary modification should be made to the used dataset. It should be customized as per the project's need or create a fresh dataset if the existing public dataset does not meet the quality criteria. Existing literature severely lacks dataset validation, where the appropriateness of the data sample should have been checked before using them to train the classification model.

5.4. Various tools/frameworks available for software defect prediction

This section explores the tools and the frameworks created in early research. Tools and frameworks are very useful as they speed up adopting the techniques proposed in the research in the real field. As per our findings, we have found fewer tools available for defect prediction. None of the existing Software Defect Prediction tools can predict defect severity, defect estimates, code references, resource allocation, and defect types. UI-based Neural Network tool (Singh and Salaria, 2013) is prepared using Levenberg–Marquardt (LM) algorithm, which takes input as various parameters with user interaction and trains the model for defect prediction. Levenberg–Marquardt (LM) algorithm is a network training function. The tool contains the appropriate buttons to perform the prediction activity. It also claims that the developed tool has high prediction accuracy. Object-Oriented metrics are used by the tool. The PROMISE repository provided the data for the dataset. Metrics used for prediction at various levels are File, Class, Component, Method, and Quantitative. The user interface for graphical representation is developed using MATLABR2011. Data is converted to numeric values from text values as a Neural Network does not work on textual values. The tool can be extended to use a different algorithm as per the work. The tool uses 13 Neurons Feed-Forward Neural Network, containing the 3-hidden layers for defect prediction. Another tool named Defect Prediction in the Software System (DePress) (Hryszko and Madeyski, 2018) is an Open-Source framework. Wroclaw University of Science and Technology worked with Capgemini and developed the tool jointly. DePress is a framework developed on top of the KNIME, an open-source data analytics platform written in JAVA. KNIME is the integration backbone for the DePress framework. The DePress framework's principal goal is to assist empirical software analysis. DePress has been built to communicate with JIRA, a Bugzilla defect tracking system with a simple configuration without writing the custom code. DePress can also connect to versioning repositories like GIT and SVN. DePress is an Open-Source project and is available on GitHub. It is open for extension and can be used for futuristic research. DePress framework is extensible and independent of language or technology. It can be deployed standalone. It is available with clear licensing rules if someone wants to use it for commercial purposes. It can also be integrated with the metrics reader like Findbug, Checkstyle, and PMD, which are static code analysis tools that can generate defect warnings in the code based on the predefined rule sets. At last, we found a GUI-based framework for better industrial adoption research (Singh and Salaria, 2013) based on Neural Networks. This work proposes to use the framework across various phases of the software development process to detect the defects. The framework is tested with the 50 real-world applications and effectively predicts the defects in a range with minimum to maximum defects. This work proposes a model specifically for the software enhancement requirements, which is generally carried out with requirement gathering, impact analysis, development, testing, user acceptance testing, and production support. A Feed-Forward Neural Network with a Sigmoid function is used to create this framework. The GUI-based tool provides easy interaction for the project managers, and this framework is implemented using a GUI-based tool developed using Matlab R2013b. Input to the frameworks are phase-wise efforts are Production efforts, Planned review efforts, Planned prevention efforts, Planned rework efforts. Based on these inputs, the tool will predict the defects in the range from minimum to maximum. The tool does not predict the discrete figure, as it can deviate more than the actual figure. Having a range of numbers for possible defects, the project manager can better plan and optimize the efforts. Table 20 shows the various Software Defect Prediction tools with their features.

5.5. Futuristic direction for software defect prediction?

Traditional methods for finding software defects can identify the limited set of defects based on executed use cases. It is possible that a few edge conditions get missed in testing and may lead to production issues. It is good to start adopting AI-based prediction tools to identify more defects based on historical data. It will surely increase the chances of figuring out more possible defects and help the software development team plan well. We have seen many researchers focus on the prediction and discuss a binary classification where the modified code is categorized into either erroneous or bug-free. This is helpful. However, more can be achieved, which generates the actionable item for the software development team to consume the predicted information and start acting on it. Knowing a particular piece of code having a defect does not give much information to the development team on what next action should be taken. Industry adoption of Software Defect Prediction approaches is quite low. Not many companies use any prediction other than static code reviews, a rule-driven framework that tries to identify the possible defects in the source code. More useful information can be predicted based on the historical datasets, which can help software development teams to improve the reliability of the produced software by predicting the defect, and also providing the information about the source code changes required, which tells what code to be modified to fix the defect, which resource can be allocated for fixing the defect, how much time it takes to fix the defect or what is the severity of the defect? Such information is also helpful information that can help better planning for the software development team and put their energy at the right place. Predicting more useful information using Artificial Intelligence techniques for the software development team can help plan and allocate resources more efficiently, detect and fix more defects, and make the delivered code bug-free. Fig. 17 depicts probable predictions.

This section explores the possible futuristic recommendations and the proposed architecture to predict the defect severity, defect estimates, code references, resource allocation, and defect types. These

Table 20
Software Defect Prediction tools/framework.

Name	Paper	Based on	User In interface	Advantage	Disadvantage	Remarks
Neural Network tool	Singh and Salaria (2013)	Levenberg–Marquardt (LM) algorithm	Matlab R2011	Using the Neural Network technique and LM algorithm gives better accuracy than a Polynomial-based Neural Network.	It supports only a single algorithm as of now.	This tool can be extended to use a different algorithm.
Defect prediction (De-Press)	Hryszko and Madeyski (2018)	KNIME, an Open-source data analytics platform	JAVA	The overall cost of the Quality Assurance work can be reduced significantly using the tool.	It supports multiple Machine Learning algorithms. Lacks sophisticated prediction models which use software process metrics, which could enhance results.	The source code is available on GitHub. https://github.com/ImpressiveCode/ic-depress
Neural Network (Feed Forward)	Vashisht et al. (2016)	Neural Network (Feed Forward) with Sigmoid function	Matlab R2013b	SDLC phases like Requirement gathering, construction, and testing showed significant improvements in early defect prediction and accuracy using this tool.	The proposed tool is unsuitable for ERP, Agile, and Production Support and needs additional efforts.	The effectiveness of the tool is tested with 50 real-world applications and found effective.

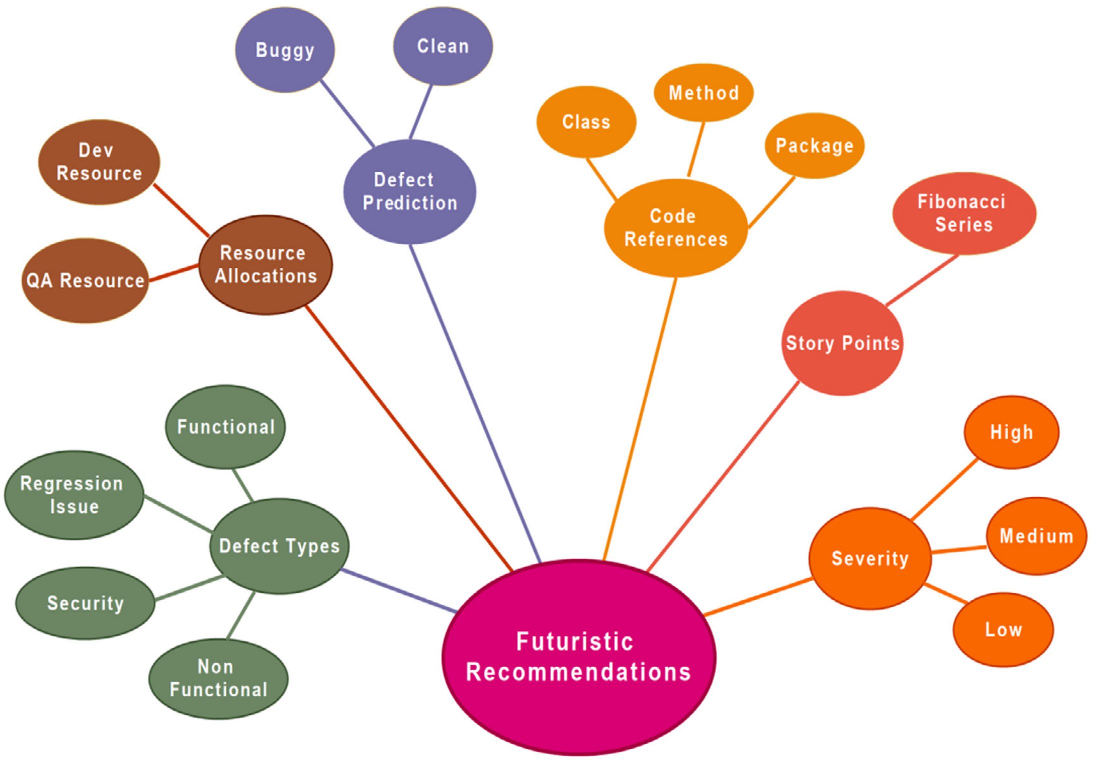

Fig. 17. Predictions for software defects.

features should be included in a user-interactive tool that the software development team can use to generate predictions. As per the survey, it is found that most of the early research is merely dealing with the binary classification of the Software Defect Prediction, which is also not easily usable in real software development work unless we provide more concrete actionable information to the software development team. It is like telling code X can have a bug that cannot be acted upon immediately. However, it is certainly helpful to focus on such priority observation. Still, this leads to quite a manual work for analysis and identification of how to consume the information and the next step? Here is a possible approach that can certainly boost the actionable task for the development team to overcome this issue. Below are the details about the proposed framework, which contains various steps required to develop a tool and predict the defect severity, defect estimates, code references, resource allocation, and defect types.

5.5.1. *High-level architecture*

Fig. 18 shows a high-level architecture flow diagram for the proposed architecture.

Step 1: Collecting dataset

Create a dataset capable of Software Defect Prediction and predict more meaningful information around defects like defect estimates, resource allocation, and references to the code for fixing the defect. Dataset can be a custom dataset or any publicly available dataset that can be further enriched to add additional features that can help predict the defect estimates, resource allocation, and the references to code for fixing the defects. Most of the public datasets used in early research do not contain the information to predict the additional defect-related details. Hence, enhancing or creating a custom dataset with all the required features is recommended. The existing public dataset is designed for binary classification and cannot be used for additional

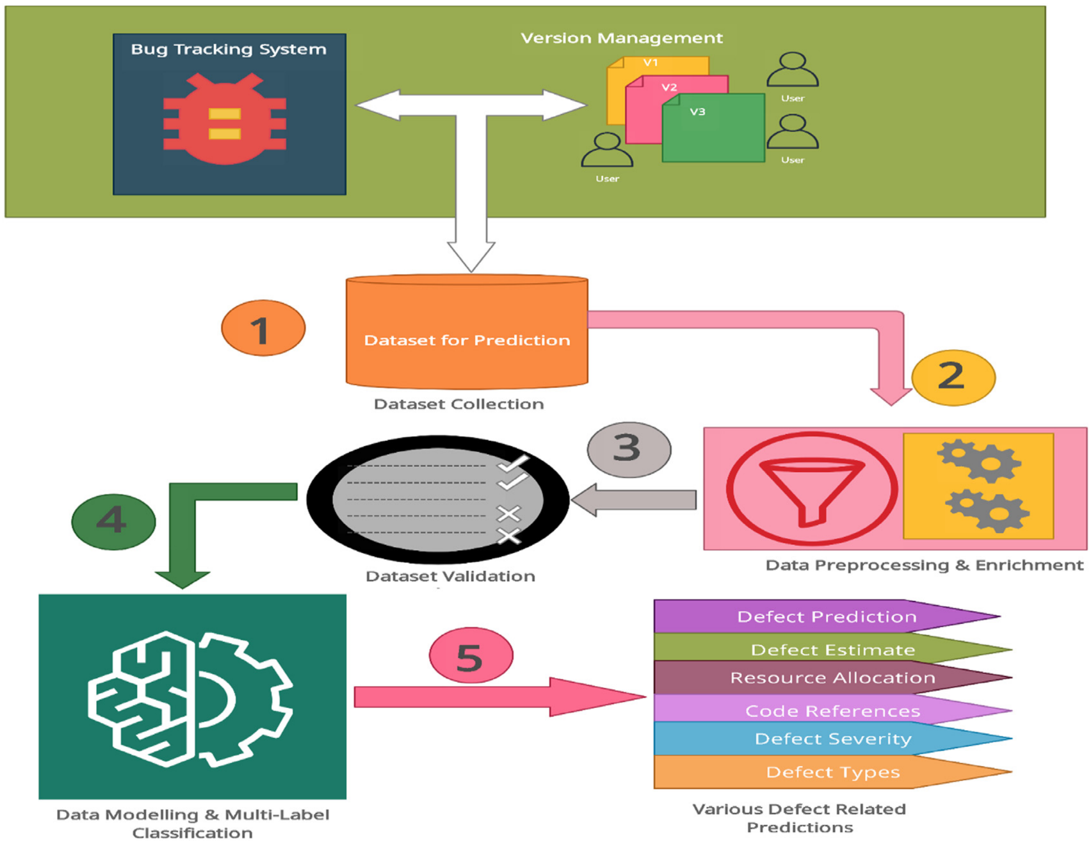

Fig. 18. High-level architectural flow.

predictions like Defect Estimates, References to the code for fixing the defect, Defect Severity, Resource Allocation, and Defect types. It is recommended to create the fresh dataset using the bug repository or enhance the existing dataset to add the relevant features for Multi-Label prediction, predicting the estimates, resource allocations, and code fix for the defects. Data can be collected from a publicly available repository like GitHub. This proposal recommends using an Open-Source repository like GitHub because they have their source code repository attached with the defect dataset. At the same time, information is available on which resource is worked on which file, the time is taken to fix the defect, and the severity of the defect. These all need to be extracted from the defect dataset and the updated reference of the code to fix the defect. Fig. 19 represents the internal service level details of the proposed architecture.

Step 2: Data enrichment

Possible ways to collect the dataset information are getting the details from a public repository like GitHub, which contains various projects with their source code. They also contain the various defects raised in different releases of those products. Having such information is very useful for dataset creation and defect prediction, estimating the time, and coming up with resource allocation and code fixes, as it is linked with the actual source code. We can also point out which source code needs to be modified to fix the defect.

Step 3: Data validation

Once the dataset has been collected, it is necessary to clean it up and remove redundant information. Once the dataset is cleaned, it is critical to know whether the collected data can predict software defects, defect estimates, resource allocation, and generate the code reference pointers for fixing the predicted defects. This step is required to perform appropriate data validations, ensuring that the collected dataset is good enough for prediction. At this stage, it is recommended to use an appropriate data validation method to check the features' relevance and predict the desired outcomes. Now we have the dataset ready. Once it is validated that this dataset is good, we can do further optimization by selecting the optimum features using features selection methods and developing the significant features that can generate a good quality learning model. Ensuring data validation helps us speed up the performance for the classification and the accuracy of the predicted results. The software defect dataset usually has the class imbalance issue, where the defect class labels are less than the class with no defects. It is recommended to pay attention to the issue of class imbalance in the newly created dataset.

Step 4: Building multi-label prediction model

Now is the time to make the actual prediction, which will require training of the classification model. Here, we can leverage the various available Artificial Intelligence techniques, specifically Deep Learning and Machine Learning algorithms, to predict defect severity, defect estimates, code references, resource allocation, and defect types. The tool can be built to support various Artificial Intelligence algorithms where users can choose the classification algorithms from a set of supported algorithms and predict the Multi-Label classification. In this case, each row of the dataset belongs to multiple labels. For example, the first label can tell it is a software defect, and the second label says this defect is of medium severity defect. Another label can predict the estimate like a high, low, medium. It can also predict the resources that can work on this particular defect. Most importantly, the code changes required can also be predicted, for example, which module or particular source class file must be modified to fix this defect. Now that defect classification and other predictions are completed, it is time to evaluate. Here we can come up with various evaluation

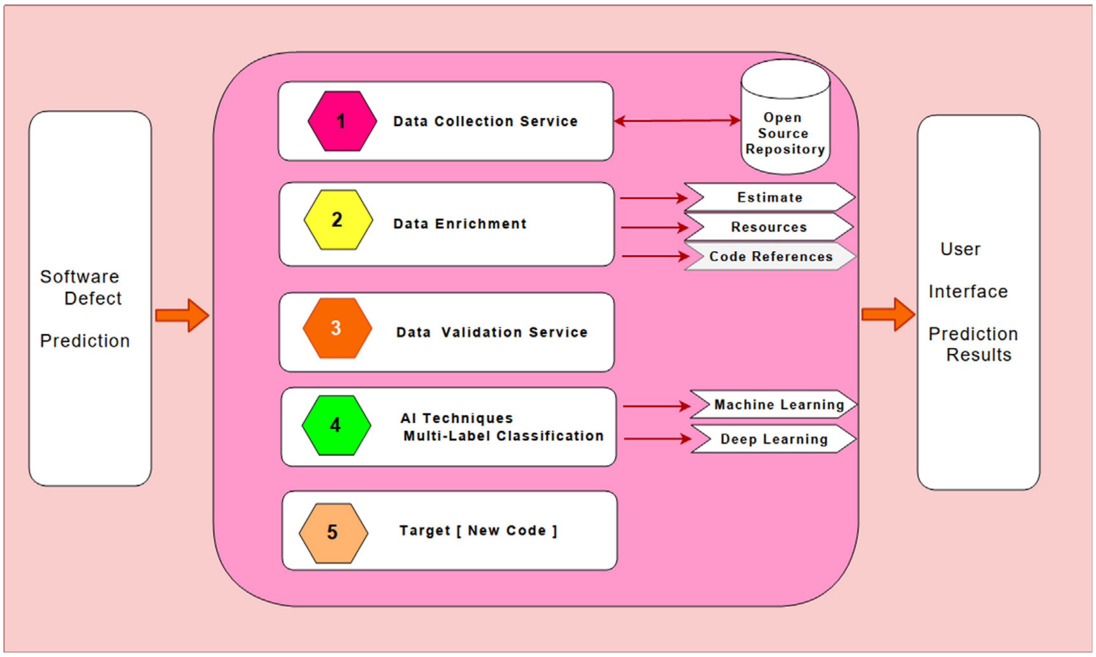

Fig. 19. Proposed high-level architecture.

strategies, and based on the Precision, Recall, F-Measure, we can check the algorithm's performance chosen for classification. The development team can choose the appropriate algorithm they would like to go with based on their context. The performance of the prediction varies as per the project, dataset, class imbalance, feature selection, etc., and it is hard to generalize which algorithm will perform better in all the possible scenarios. Hence, the tool must support multiple Artificial Intelligence algorithms that can be chosen for Multi-Label classification as per the requirement of the software development team.

Step 5: Predicting the new code

All of the steps mentioned above are aggregated into a common single-user interactive tool that can generate multi-label predictions for the newly developed software code like defect severity, defect estimates, code references, resource allocation, defect types, etc. The interactive user interface should display various predictions related to software defects.

5.5.2. Additional metrics

According to the survey, most researchers preferred to use publicly available datasets. Most of the studies employed the metrics to train the model: Process related metrics, Attributes taken from the previously filed defects, Source code metrics in general, Object-oriented metrics. Process-related metrics are about the process attributes like the number of developers involved, the number of code lines changed, the file's version number, file refactoring, etc. Apart from this, another set of attributes available in the datasets is source code metrics. In this category, most researchers have used Object-Oriented metrics like depth of inheritance, class coupling, nested looping, if-else-ladder, number of public methods, etc. These are the appropriate indicators that usually indicate the complexity of the code. If code is complex, it has a high probability of having defects.

Proposed additional metrics from previously filed defects

Previous defects are the key data element in Software Defect Prediction considering Supervised Machine Learning. Usually, this data can predict the defects in the same project or the cross projects. Many public dataset repositories like NASA and PROMISE are available, which are used for binary defect classification. However, additional metrics will be required to help us predict the defect severity, defect estimates, code references, resource allocation, and defect types, which is vital for the software development team. Additional attributes used from the previous defects can significantly improve the classification and develop a more actionable outcome than mere classification. Some of the significant attributes are:

1. What resources have worked on the defects?
2. Which use case impacted?
3. How much time did it take to fix and test the defects?
4. Did it cause regression?
5. What was the severity of the defect?
6. What code changes have fixed the defects?
7. What is the method's name where the code needs to be fixed?
8. What is the number of parameters passed to the method?
9. What is the length of the method where the code changes are done to fix the defect?
10. What is the type of defect?
11. Are there any side effects or regression?

Some of the finest questions can be answered using the data available from the previous defects. Such rich data can help predict more valuable information of correlated defects, which probably need to be manually investigated if Artificial Intelligence techniques are not leveraged. Although the features mentioned in the additional metrics list are critically important, none of the public datasets covers such a rich set of features. The reason for the creation of the available public dataset was performing the binary classification to predict if the code is defective or not? Hence it is not fair to expect the existing public dataset to have a rich feature set for predicting additional information, which was not intended at the time of dataset creation. If we need to provide additional details like defect severity, defect estimates, code references, resource allocation, and defect types, we need to dig down the defect data source. Open-Source projects can be explored to precisely bring out such significant attributes for every defect. Lack of features in the existing dataset suggests we should create a custom dataset with a rich feature set to produce more actionable outcomes.

6. Discussion

This section discusses the outcome of the Systematic Literature Review, which is conducted by selecting 146 research studies. We have explored relevant literature related to the formulated research questions, and based on the review and observations, we can get the answers to the formulated research questions.

RQ1: What are the various approaches for finding defects in the newly developed code?

As per the survey, there are two observations, and we have categorized those observations into two categories: Defect detection and Prediction techniques. The popular way of identifying and fixing defects in the industry is using defect detection. Once the code is ready, or the development phases are over, the Quality Assurance engineer starts manual testing. Few tests can be automated and run with every build using the continuous build system. Continuous integration (Philip et al., 2019; Bertolino et al., 2020) and building system keep identifying the defects after new code development. Similarly, if different teams integrate different parts of the code, the Quality Assurance engineer does manual integration testing. The main drawback of the detection approach is that it is time-consuming, and more effort is required to identify defects. Because of the short release cycle and time to the market requirement, it is nearly impossible to spend a lot of time in defect identification and fixing those defects. There is a huge requirement in the software industry to identify the priority areas that the Quality Assurance team should focus on to identify defects and ensure that the product can be released in minimum time, considering the market situation or the peer pressure. As per the survey, we have identified that manual testing, automation testing, and unit testing do not identify many defects as executing the static steps or the pre-defined automated test scripts. They follow certain test steps and do not deviate much with predefined steps, which have already been identified as a test case. There may be many edge conditions that might get ignored while doing detection-based testing. It also does not consider historical information like integrating two components. For example, Component C1 is integrated with Component C2., There may be different kinds of issues in the production environment that did not get covered in Unit testing or manual testing, so this kind of information may get ignored. Leveraging historical data of the previous release or the historical data from similar projects can help predict more defects. This SLR does not recommend removing the detection-based techniques. However, it is recommended to include the prediction-based techniques using the Artificial Intelligence approach, specifically Machine Learning and Deep Learning to predict more defects, so it can figure out the area which needs the focus and accordingly the resources can be distributed for the priority use cases with high significance or high severity. This approach will ensure very good utilization of the resources, and it helps develop a product with fewer defects in the critical use cases. Defect prediction saves time and money and provides an optimum quality product. The early researchers have used Machine Learning and Deep Learning techniques. They have identified algorithms that work great with different datasets in different scenarios. However, it is difficult to generalize the best algorithm as the context of the software project is an important factor to be considered. As per the survey, Machine Learning and Deep Learning algorithms have proven that these techniques are suitable for predicting defects. This SLR proposes to extend this binary classification capability to Multi-Label classification, using the Machine Learning and Deep Learning techniques, with dataset customization having extra features extracted from the defect dataset version management system, to provide more meaningful information to the software development team to make the product more robust with a smaller number of defects.

RQ2: What are the available datasets, and are they good enough for predicting various actionable tasks related to defects?

The majority of the researchers have used the standard dataset NASA, PROMISE, AEEEM datasets. Few researchers have created their datasets using Open-Source projects like GitHub source code repository to generate the high-quality dataset for binary classification. According to the survey, most of the existing datasets have class imbalance issues and high dimensionality issues. Data pre-processing is important and must be included to make the prediction more accurate. Few researchers considered datasets-related issues like high dimensionality class imbalance before training the model using the dataset. The existing standard dataset is good enough for binary classification, considering if the class imbalance issue is taken care of before usage. Although the existing datasets are suitable for binary classification, they do not contain enough features for predicting the Defect estimates, Resources that can work on the defect, Code that needs to be updated for fixing the defect, Type of the defects, and the severity of the defect. Hence, if we want to predict more information related to software defects, it is recommended to create a new dataset or a customized dataset with enough features to predict the additional information related to software defects. Open GitHub repository is one of the significant resources which could be utilized for creating the training dataset as it contains:

1. Updated source code with Version management, whether new or existing code
2. Historical defects
3. Resources worked on the defects
4. The severity of the defects
5. References to the code which is modified to fix the defects
6. Time took for fixing the defects

The information mentioned above is significant. Utilizing this information for planning and execution in software development projects will make a big difference in the software delivery quality. It is recommended to extract this information from the publicly available Open-Source project and develop a customized dataset that is good enough to predict the additional defect-related useful information for the software defects. Considering the research questions, it is evident that if we need to predict defect severity, defect estimates, code references, resource allocation, and defect types, we need to have a dataset trained for Multi-Label classification with the mentioned additional features.

RQ3: What are the available data validation techniques to ensure that training data is appropriate for Software Defect Prediction modeling?

The suitability of the sample for building the classification model is assessed by dataset validation. If the selected samples are not good quality, whatever algorithm is used does not provide good results. As per the survey, one of the methods which can be used to check the appropriateness of the dataset is Cross Fold Validation (Taek et al., 2016; Yu et al., 2018). Cross-Fold Validation, also known as K-Fold Cross-Validation, is a valuable approach for removing overfitting from training dataset results. Overfitting (Dong et al., 2018) is handled with Cross-Fold validation with a K value of 5. When the dataset is small, over-fitting is also caused by choosing too many parameters. Appropriate feature selection and careful selection of the Cross-Fold Validation are a few of the significant validation steps that can improve the prediction quality of the classification model. Although many non-parametric tests were conducted to validate the difference in classification results, the statistical test for sample validation is entirely missing. The quality of the dataset is very important because it defines the quality of the prediction. As per the findings, very few studies included the data validation steps before using the dataset for Software Defect Prediction. Suppose we want to predict additional attributes for the software defects, like resource allocation, code references for fixing the defect, or estimating the defect. In that case, we need to ensure that the collected data is of good quality. Data validation is essential as all the defect-related information might not have been captured by the

Quality Assurance engineer, and missing data or unrelated data used for training may lead to a poor prediction model.

RQ4: What are the various tools/frameworks available for Software Defect Prediction?

As per the survey, very few tools are designed explicitly for Software Defect Prediction. Two tools are Neural Network-based, and another tool named De-Press is based on the KNIME framework. De-Press is jointly developed by Wroclaw University of Science and Technology and Cap Gemini. The user interfaces for the DePress are designed using JAVA language, whereas the Neural Network-based user interface is designed using MATLAB. These tools can do the Software Defect Prediction using binary classification by predicting whether the newly developed code has a bug or not. DePress tool is internally using the KNIME framework, an Open-Source platform for data analytics. It has a lot of built-in algorithms for Machine Learning and Deep Learning. These tools can also be extended for further improvements as per the study. Tools based on the Neural Networks have little difference in their implementation. One is based on the LM algorithm, and another uses the Feed-Forward Neural Network with Sigmoid function. A detailed comparison is provided in Table 20. Apart from the dedicated tools for Software Defect Prediction, other Open-Source tools named WEKA are used by many researchers to do various analyzes related to datasets, sample selection, Cross-Validation, and classification using various inbuilt supported Machines and Deep Learning algorithms. Considering the limited availability of the dedicated software prediction tools, specifically the one which can predict more information about software defects like defect severity, defect estimates, code references, resource allocation, and defect types, it is recommended to build the tool capable of predicting such additional details. It should provide the proper user interface for performing various operations that take the user's input and perform multiple steps as recommended in the proposed architecture in Fig. 19. Such tools can help the industry to adopt and use the prediction capabilities of Artificial Intelligence techniques. It eases the software development work by smoothing out various complex steps to set up the prediction environment.

RQ5: What is the possible futuristic direction for Software Defect Prediction, and what can be predicted for software defects to enhance the standard of the delivered software product?

Based on the gap identified in various research studies and the current trend used in the software industry to find defects, it is required to pay attention to some of the important areas of Software Defect Predictions given below:

Usage of artificial intelligence techniques. As per the survey, it is found that selecting the legacy approach, which is detection-based, can only detect a few defects. In contrast, prediction approaches can be beneficial in predicting many possible defects using the historical dataset. Hence one of the recommendations for a futuristic approach is to use Artificial Intelligence techniques, specifically Deep Learning techniques that can very well scale with large datasets and generate prediction results for the software defects with better confidence.

Need for rich feature prediction dataset. It is also recommended that instead of using the available dataset, which lacks the adequate features required to predict defect severity, defect estimates, code references, resource allocation, and defect types, we can build a custom dataset. Developing a dataset with adequate features is good if we need to predict these details. As per the survey, it is found that a public repository like GitHub is quite helpful for such an approach because it has linked the defect dataset with the source code version repository, which maintains the code changes. It also has information about the resources, who has worked on the defects and the time taken to fix the defects, and several other details available in the defect dataset. So it is a good idea to choose an Open-Source project and develop a customized dataset as per the project's need for futuristic recommendations.

Data validation. It is observed that early researchers have not taken enough care of the data validation before using the dataset for training the model, as there are very few studies that discuss the data validation on the dataset. It is critical to ensure the quality of the dataset, ensuring prediction quality. It is required to validate the data, specifically the interrelationship of the data, whether the attributes and features can predict the defects, or the dataset has adequate features for predictions. One must verify the dataset before using them in the training samples. As per the survey, we have seen very few techniques referred for the data validations like k-Fold Cross-Validation. Hence, it is recommended to explore more data validation to ensure that we have very good quality data.

Software prediction tools. As per the survey, it is observed that there are very few tools available for predicting software defects. Less industry adoption could be due to the inherent complexity and difficulty of executing various required steps for Software Defect Prediction. Setting up the prediction environment for the newly developed code is time-consuming and requires Machine Learning or Deep Learning knowledge. Many steps are involved in developing a sound Software Defect Prediction system, like selecting the samples and data validation. Existing tools are capable of only defect prediction. Hence, it is recommended to develop a tool that automates various steps and can contribute more to planning and execution for the software development team by predicting defect severity, defect estimates, code references, resource allocation, and defect types. It is recommended to develop a dedicated tool that performs most of the steps in a user interactive way and performs by taking the input from the user, finally providing the prediction results on various aspects of the software defects. This SLR proposes the high-level architecture with various steps for creating a tool dedicated to predicting software defects and other attributes related to defects, as shown in Fig. 19.

7. Limitations of the study

The presented Systematic Literature Review critically analyzed the recent trends in Software Defect Prediction and performed the Comparative analysis across the multiple approaches. The SLR examines the available techniques and the challenges of exploiting existing datasets, data validation methods utilized by early researchers, and existing Software Defect Prediction systems. According to the research, extending Software Defect Prediction from Binary classification to Multi-Label classification is a futuristic trend for enhancing software product quality. Here are the possible limitations in the presented SLR:

1. Systematic Literature Review is conducted by a well-defined search query executed on good quality research databases. However, there is a possibility of missing a few of the research that might not be retrieved with the defined criteria.
2. Retrieved articles are further scrutinized to focus on the quality research papers having accurate references and the relevance to the desired research goals, which might cause losing some relevant information.
3. Most early research centered on using binary classification to solve the prediction quality, performance, and reliability for predicting software defects. Early research did not focus on prediction for defect severity, defect estimates, code references, resource allocation, and defect types. Hence, adequate information is not available in the existing literature for Multi-Label predictions and needs more investigation for Multi-Label prediction for software defects.
4. Because different studies used diverse sets of algorithms and datasets, it is difficult to generalize the best Machine Learning or Deep Learning techniques.

The proposed architecture for developing a dedicated Software Defect Prediction tool is still in the planning stage. As a result, experimental results are beyond the scope of the presented literature, but they still provide a clear, futuristic direction and approach for developing various Software Defect Predictions.

Table 21
Public datasets referred in earlier research.

Database	Papers
CM1	(Catal, 2014; Wahono and Herman, 2014)
JM1	(Catal, 2014; Peng et al., 2009)
KC1	(Catal, 2014; Wahono and Herman, 2014)
KC2	(Catal, 2014; Zhang et al., 2010)
KC3	(Gray et al., 2009; Peng et al., 2009)
KC4	(Gray et al., 2009; Peng et al., 2009)
MC1	(Gray et al., 2009; Peng et al., 2009)
MC2	(Gray et al., 2009; Wahono and Herman, 2014)
MW1	(Peng et al., 2009; Wahono and Herman, 2014)
PC1	(Catal, 2014; Wahono and Herman, 2014)
PC2	(Hall et al., 2011; Wahono and Herman, 2014)
PC3	(Peng et al., 2009; Wahono and Herman, 2014)
PC4	(Peng et al., 2009; Wahono and Herman, 2014)
PC5	(Gray et al., 2009; Sun et al., 2012)
ECLIPSE 3.0	(Li et al., 2012; Zhang et al., 2010)
Eclipse 2.0	(Li et al., 2012)
JDT.Core	(Li et al., 2012)
SWT	(Li et al., 2012, 2018)
Xalan	(He et al., 2015; Qiu et al., 2019)
EQ	(Jing et al., 2016; Qiu et al., 2019)
JDT	(Jing et al., 2016; Taek et al., 2016)
LC	(Jing et al., 2016; Qiu et al., 2019)
JIRA DB Derby, Groovy, HBase, Hive, JRuby, ActiveMQ	(Yatish et al., 2019)

Table 22
Algorithms grouped based on similarity.

Algorithms	Type	References
Bayesian Algorithms	Bayesian Network	(Zhou et al., 2014; Okutan and Yıldız, 2014)
	Naive Bayes	(Ji et al., 2019; Zhang et al., 2015)
Artificial Neural Network	Radial Basis Function	(Ali et al., 2020; Raghava et al., 2019)
	Label propagation	(Zhang et al., 2017)
	Nonnegative sparse graph-based label propagation (NSGLP)	(Zhang et al., 2017)
	Multilayer perceptron	(Gao et al., 2015b; Zhang et al., 2015)
Clustering Algorithms	K-means	(Yadav and Yadav, 2015; Raghava et al., 2019)
Decision Tree Algorithms	C4.5	(Zhang et al., 2015; Peng et al., 2009)
	Decision forest	(Naseem et al., 2020; Siers and Islam, 2018b)
	CART	(Chen et al., 2019b)
Deep Learning Algorithms	Deep Belief Network	(Hoang et al., 2019; Liang et al., 2019)
	Convolutional Neural Networks	(Phan et al., 2018; Qiu et al., 2019)
	Tree-based Convolutional Neural Networks (TBCNN)	(Qiu et al., 2019)
	Recurrent Neural Networks	(Fan et al., 2019)
	Long Short Term Memory Network	(Hoa et al., 2019; Liang et al., 2019)
Dimensionality Reduction	Principal Component Analysis	(He et al., 2019; Ji and Huang, 2018)
	Kernel Principal Component	(Ren et al., 2014; Xu et al., 2019)
	Linear Discriminant Analysis	(Ma et al., 2014)
Instance-based	Support vector machines	(Catal, 2014; Zhang et al., 2015)
	K-Nearest Neighborhood	(Okutan and Yildiz, 2016; Yu et al., 2017b)
	Weighted Extreme Learning Machine	(Xu et al., 2019; Zheng et al., 2020)
Regression Algorithms	Linear regression	(Kakkar et al., 2021)
	Logistic Regression	(Hall et al., 2011; Zhang et al., 2015)
Ensemble Algorithms	Adaboost	(He et al., 2019; Khoshgoftaar et al., 2015)
	Boosting	(Zhang et al., 2015, 2018b)
	Bagging	(Zhang et al., 2015; Raghava et al., 2019)
	Random Forest (RF)	(Catolino et al., 2019; Zhang et al., 2015)
	Stacking	(Akour et al., 2017; Saifan and Abu-wardih, 2020)
	Over-bagging	(Mousavi et al., 2018)

8. Conclusion

This systematic literature review explores the recent trends in Software Defect Prediction using Artificial Intelligence techniques and tries to identify the research gaps and further enhance Software Defect Prediction opportunities. The SLR adheres to the guidelines proposed by Kitchenham and Charters (Kitchenham et al., 2009) while conducting the literature search. To choose high-quality research papers for a literature review, inclusion and exclusion criteria and quality assessment criteria are clearly stated. 146 publications were identified and selected for analysis to address the formulated research questions. The majority of the selected studies are from the previous ten years. The SLR does the critical analysis for Software Defect Prediction research as stated below:

1. Identified the various approaches for finding the software defects and any additional information related to the software defects that the software development team can utilize.
2. Identified dataset problems for Software Defect Prediction, such as class imbalance and insufficient features, limit prediction to binary classification.

Table 23
Significant conferences and journals that were referred to in this SLR.

No.	Journals/Conferences	No	Journals/Conferences
1.	IEEE Transactions on Software Engineering	11	Information and Software Technology
2.	IEEE Access	12	IET Software
3.	IEEE Transactions on Reliability	13	International Conference on Software Engineering International Workshop on Machine-Learning Techniques for Software-Quality Evaluation, Co-located with ESEC/FSE 2020
4.	IEEE Transactions on Systems, Man and Cybernetics-Part C: Applications and Reviews	14	ESEC/FSE ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering
5.	ACM Transactions on Software Engineering and Methodology (TOSEM)	15	International Conference on Machine Learning and Applications, ICMLA
6.	International Journal of Software Engineering and Knowledge Engineering	16	International Conference on Machine Learning and Applications, ICMLA 2010
7.	International Journal of Open Source Software and Processes	17	International Conference on Tools with Artificial Intelligence, ICTAI
8.	Journal of Systems and Software	18	International Symposium on Empirical Software Engineering and Measurement, ESEM
9.	Automated Software Engineering	19	Software Quality Journal
10.	Empirical Software Engineering	20	Engineering Applications of Artificial Intelligence (EAAI)

Table 24
Abbreviation of the terms referred to in this SLR.

PCA	Principal Component Analysis	DT	Decision Tree	KNN	K-Nearest Neighbors
RF	Random Forest	ANN	Artificial Neural Network	LR	Logistic Regression
NB	Naive Bayes	LSTM	Long Short-Term Memory	CPDP	Cross Project Defect Prediction
SVM	Support Vector Machine	PPM	Proposed Process Metrics	WPDP	Within Project Defect Prediction
NN	Neural Network	xGBTree	Optimized Extreme Gradient Boosting Trees	MLP	Multilayer Perceptron
DBN	Deep Belief Network	GBM	Gradient Boosting Machine	DL	Deep Learning
CNN	Convolutional Neural Network	MDT	Manifold Detection	DePress	Defect Prediction in the Software System
SLR	Systematic Literature Review	TCNN	Transfer Convolutional Neural Network	ML	Machine Learning
RNN	Recurrent Neural Network	ISDA	Improved Subclass Discriminant Analysis	UI	User Interface
GUI	Graphical User Interface				

3. The necessity to investigate various data validation strategies used to verify the dataset quality for defect prediction was identified, as it had not received enough attention in the previous research.
4. It is identified that it is required to enhance the existing dataset or create a dataset from scratch, including the additional feature set that can help us determine the defect severity, defect estimates, code references, resource allocation, and defect types.
5. Identify the need to develop a dedicated tool for Software Defect Prediction and predict more information related to defects. There are very few tools available and limited to only defect prediction using binary classification. The creation of user interface-based tools enhances the industry adoption of the prediction-based approaches.

According to the survey's findings, several sorts of studies focus on Artificial Intelligence-based techniques such as Machine Learning and Deep Learning for predicting software defects because of their effectiveness in delivering correct results with high confidence. This SLR recommended a futuristic direction for Software Defect Prediction and presented a high-level architecture diagram that can be used to develop a framework or the tool, which is capable of not only Software Defect Prediction but also able to predict the defect severity, defect estimates, code references, resource allocation, and defect types. This information is significant for any software development team to fix the defects quickly with great confidence and allocate the Quality Assurance engineer on the priority task rather than focusing on the area where we may see fewer defects. The findings of this study can help improve the quality of software products that incur financial losses, increased time to market, and customer unhappiness due to a high number of defects in the final product. A prediction-based system can predict more defects than the traditional approach. The predicted defect can be fixed before delivery, which otherwise occurs in the actual production environment. Hence, it reduces the cost and the time of software development by getting the proper visibility and priority on various development-related tasks and helps deliver robust software products by minimizing defects.

9. Future work and opportunities

In the discipline of software development, specifically finding and fixing software defects, there is a lot of potentials. As shown in Fig. 17, we can see that we can predict more useful information related to software defects rather than just a binary prediction for Defective versus Non-Defective code. The adequate focus needs to be given on predicting defect severity like high, medium, or low, and estimation can also be predicted for the specific defect using the Fibonacci Series in terms of story points or high, low, medium. From the historical defect dataset and the Code Version management system, possible code changes can be predicted for the specific defects by classifying the defect to a particular module, packages. Similarly, based on the historical data, resource allocation for the defect can also be predicted based on the attributes like for similar defects which resource from the team had worked in the past and have good experience on the specific codebase. We may also forecast or classify the defect's category, whether functional or Non-Functional issue. Is this a security flaw? Or is there a problem as a result of a regression? These are the significant pieces of information that will be very helpful to the software development team to plan their work for the appropriate stories. Accordingly, resource allocation can be done in a more optimized way. Prediction-based techniques can identify more defects than actual manual testing, which is always limited due to cost issues, which will help reduce the number of unidentified defects in the production environment. This SLR does not recommend replacing the traditional approaches of defect fixing with Prediction-based. Prediction-based approaches can add a lot of value if used with hybrid implementation and having reliable prediction knowledge can be very helpful in better planning. Here are some of the recommendations made by this study:

1. Identify appropriate datasets or create a custom dataset with adequate features to leverage for prediction for more helpful information other than just defects.
2. Ensure that the collected dataset should have proper labels for the new prediction classes. This will be required for training the classification model.
3. Appropriate data validation procedures must be applied to ensure that the data is of acceptable quality and can produce credible predictions for the various categories of Software Defect Prediction.
4. The collected dataset should have rich features and should be helpful to predict more information about the software defect. We need to explore the appropriate Artificial Intelligence techniques useful for Multi-Label classification.

Industry adoption of Software Defect Prediction is relatively low as it needs a complex setup and efforts to execute the prediction. Various predictions can be aggregated in a dedicated Software Defect Prediction tool, which can also predict the most helpful information about the software defect. This tool should be an interactive user tool and allow the user to complete various steps required for the defect prediction.

Miscellaneous

Table 21 lists the literature review references for the datasets. Based on their functioning and similarity of their operations, algorithms grouped are shown in Table 22. Table 23 presents some of the significant conferences and journals referred to in this SLR. Abbreviations that are used in this SLR are listed in Table 24.

CRediT authorship contribution statement

Jalaj Pachouly: Conceptualization, Methodology, Writing – first and second draft, Formal analysis, Investigation, Visualization. **Swati Ahirrao:** Conceptualization, Methodology, Data curation, Writing – first and second draft, Formal analysis, Investigation, Supervision. **Ketan Kotecha:** Resources, Formal analysis, Validation, Funding acquisition, Project administration, Supervision. **Ganeshsree Selvachandran:** Writing – review & editing, Formal analysis, Validation, Software. **Ajith Abraham:** Writing – review & editing; Validation, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank the Editors and the anonymous reviewers for their valuable comments and suggestions. The authors would like to gratefully acknowledge the financial assistance received from the Ministry of Higher Education, Malaysia under grant no. FRGS/1/2020/STG06/UCSI/02/1.

Ethical approval

This article does not contain any studies with human participants or animals performed by any of the authors.

References

Abdulshaheed, Mohamed, et al., 2019a. Mining historical software testing outcomes to predict future results. Compusoft 8 (12), 3525–3529.

Agrawal, Amritanshu, Menzies, Tim, 2018. Is better data better than better data miners? In: 2018 IEEE/ACM 40th International Conference on Software Engineering. ICSE, IEEE.

Akour, Mohammed, Alsmadi, Izzat, Alazzam, Iyad, 2017. Software fault proneness prediction: a comparative study between bagging, boosting, and stacking ensemble and base learner methods. Int. J. Data Anal. Tech. Strateg. 9 (1), 1–16.

Ali, Umair, et al., 2020. Software defect prediction using variant based ensemble learning and feature selection techniques. Int. J. Modern Educ. Comput. Sci. 12 (5).

Anon, 2013. Cambridge university study states software bugs cost economy dollar 312 billion per year. Retrieved from Financial Content: Cambridge University study states software bugs cost economy dollar 312 billion per year.

Anon, 2018a. Software fail watch 5th edition. Retrieved from https://www.tricentis.com/resources/software-fail-watch-5th-edition/.

Anon, 2018b. The cost of poor-quality software in the us: a 2018 report (2018, september 26). Retrieved from: https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf.

Bashir, Kamal, Li, Tianrui, Yohannese, Chubato Wondaferaw, 2018. An empirical study for enhanced software defect prediction using a learning-based framework. Int. J. Comput. Intell. Syst. 12 (1), 282.

Bertolino, Antonia, et al., 2020. Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.

Bowes, David, Hall, Tracy, Petrić, Jean, 2018. Software defect prediction: do different classifiers find the same defects? Softw. Qual. J. 26 (2), 525–552.

Cabral, George G., et al., 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE.

Catal, Cagatay, 2014. A comparison of semi-supervised classification approaches for software defect prediction. J. Intell. Syst. 23 (1), 75–82.

Catal, Cagatay, Diri, Banu, 2009. A systematic review of software fault prediction studies. Expert Syst. Appl. 36 (4), 7346–7354.

Catolino, Gemma, Nucci, Dario Di, Ferrucci, Filomena, 2019. Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE.

Caulo, Maria, Scanniello, Giuseppe, 2020. A taxonomy of metrics for software fault prediction. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications. SEAA, IEEE.

Chen, Jinyin, et al., 2019a. Multiview transfer learning for software defect prediction. IEEE Access 7, 8901–8916.

Chen, Jianfeng, et al., 2019b. Predicting breakdowns in cloud services (with SPIKE). In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

Chen, Xiang, et al., 2019c. Software defect number prediction: Unsupervised vs supervised methods. Inf. Softw. Technol. 106, 161–181.

Chen, Jinyin, et al., 2020. Software visualization and deep transfer learning for effective software defect prediction. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.

Choeikiwong, Teerawit, Vateekul, Peerapon, 2016. Two stage model to detect and rank software defects on imbalanced and scarcity data sets. IAENG Int. J. Comput. Sci. 43 (3).

Cruz, Ana Erika Camargo, Ochimizu, Koichiro, 2009. Towards logistic regression models for predicting fault-prone code across software projects. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE.

Cui, Di, et al., 2019. Investigating the impact of multiple dependency structures on software defects. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE.

D'Ambros, Marco, Lanza, Michele, Robbes, Romain, 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Softw. Eng. 17 (4), 531–577.

Di Mario, Mauro, et al., 2021. Supervised feature selection techniques in network intrusion detection: A critical review. Eng. Appl. Artif. Intell. 101, 104216.

Dong, Feng, et al., 2018. Defect prediction in android binary executables using deep neural network. Wirel. Pers. Commun. 102 (3), 2261–2285.

Elmishali, Amir, Stern, Roni, Kalech, Meir, 2018. An artificial intelligence paradigm for troubleshooting software bugs. Eng. Appl. Artif. Intell. 69, 147–156.

Fan, Guisheng, et al., 2019. Software defect prediction via attention-based recurrent neural network. Sci. Program. 2019.

Felix, Ebubeogu Amarachukwu, Lee, Sai Peck, 2017. Integrated approach to software defect prediction. IEEE Access 5, 21524–21547.

Gao, Kehan, Khoshgoftaar, Taghi M., 2015. Assessments of feature selection techniques with respect to data sampling for highly imbalanced software measurement data. Int. J. Reliab. Qual. Saf. Eng. 22 (02), 1550010.

Gao, Kehan, Khoshgoftaar, Taghi M., Napolitano, Amri, 2015a. Aggregating data sampling with feature subset selection to address skewed software defect data. Int. J. Softw. Eng. Knowl. Eng. 25 (09n10), 1531–1550.

Gao, Kehan, Khoshgoftaar, Taghi M., Napolitano, Amri, 2015b. Investigating two approaches for adding feature ranking to sampled ensemble learning for software quality estimation. Int. J. Softw. Eng. Knowl. Eng. 25 (01), 115–146.

Gao, Kehan, Khoshgoftaar, Taghi M., Wald, Randall, 2014. The use of under-and oversampling within ensemble feature selection and classification for software quality prediction. Int. J. Reliab. Qual. Saf. Eng. 21 (01), 1450004.

Ghosh, Soumi, Rana, Ajay, Kansal, Vineet, 2018. A nonlinear manifold detection based model for software defect prediction. Procedia Comput. Sci. 132, 581–594.

Gong, Lina, Jiang, Shujuan, Jiang, Li, 2019. Tackling class imbalance problem in software defect prediction through cluster-based over-sampling with filtering. IEEE Access 7, 145725-145737.

Gray, David, et al., 2009. Using the support vector machine as a classification method for software defect prediction with static code metrics. In: International Conference on Engineering Applications of Neural Networks. Springer, Berlin, Heidelberg.

Hall, Tracy, et al., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.

He, Peng, et al., 2015. An empirical study on software defect prediction with a simplified metric set. Inf. Softw. Technol. 59, 170–190.

He, Haitao, et al., 2019. Ensemble multiboost based on ripper classifier for prediction of imbalanced software defect data. IEEE Access 7, 110333-110343.

Hoa, Khanh Dam, et al., 2019. Lessons learned from using a deep tree-based model for software defect prediction in practice. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE.

Hoang, Thong, et al., 2019. DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE.

Hosseini, Seyedrebvar, Turhan, Burak, Gunarathna, Dimuthu, 2017. A systematic literature review and meta-analysis on cross project defect prediction. IEEE Trans. Softw. Eng. 45 (2), 111–147.

Hryszko, Jaroslaw, Madeyski, Lech, 2018. Cost effectiveness of software defect prediction in an industrial project. Found. Comput. Decis. Sci. 43 (1), 7–35.

Huda, Shamsul, et al., 2017. A framework for software defect prediction and metric selection. IEEE Access 6, 2844–2858.

Huda, Shamsul, et al., 2018. An ensemble oversampling model for class imbalance problem in software defect prediction. IEEE Access 6, 24184–24195.

Jakhar, Amit Kumar, Rajnish, Kumar, 2018. Software fault prediction with data mining techniques by using feature selection based models. Int. J. Electr. Eng. Inf. 10 (3).

Jayanthi, R., Florence, Lilly, 2019. Software defect prediction techniques using metrics based on neural network classifier. Cluster Comput. 22 (1), 77–88.

Ji, Haijin, Huang, Song, 2018. Kernel entropy component analysis with nongreedy L1-norm maximization. Comput. Intell. Neurosci. 2018.

Ji, Haijin, et al., 2018. A two-stage feature weighting method for naive Bayes and its application in software defect prediction. Int. J. Perform. Eng. 14 (7), 1468.

Ji, Haijin, et al., 2019. A new weighted naive Bayes method based on information diffusion for software defect prediction. Softw. Qual. J. 27 (3), 923–968.

Jing, Xiao-Yuan, et al., 2016. An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems. IEEE Trans. Softw. Eng. 43 (4), 321–339.

Kakkar, Misha, et al., 2021. Combining data preprocessing methods with imputation techniques for software defect prediction. In: Research Anthology on Recent Trends, Tools, and Implications of Computer Programming. IGI Global, pp. 1792–1811.

Khamis, Mohamed A., Gomaa, Walid, 2015. Comparative assessment of machine-learning scoring functions on PDBbind 2013. Eng. Appl. Artif. Intell. 45, 136–151.

Khoshgoftaar, Taghi M., Gao, Kehan, 2009. Feature selection with imbalanced data for software defect prediction. In: 2009 International Conference on Machine Learning and Applications. IEEE.

Khoshgoftaar, Taghi M., Gao, Kehan, Seliya, Naeem, 2010. Attribute selection and imbalanced data: Problems in software defect prediction. In: 2010 22nd IEEE International Conference on Tools with Artificial Intelligence, Vol. 1. IEEE.

Khoshgoftaar, Taghi M., et al., 2015. Comparing feature selection techniques for software quality estimation using data-sampling-based boosting algorithms. Int. J. Reliab. Qual. Saf. Eng. 22 (03), 1550013.

Khuat, Thanh Tung, Le, My Hanh, 2019. Ensemble learning for software fault prediction problem with imbalanced data. Int. J. Electr. Comput. Eng. 9 (4), 3241.

Kitchenham, Barbara, et al., 2009. Systematic literature reviews in software engineering–a systematic literature review. Inf. Softw. Technol. 51 (1), 7–15.

Laradji, Issam H., Alshayeb, Mohammad, Ghouti, Lahouari, 2015. Software defect prediction using ensemble learning on selected features. Inf. Softw. Technol. 58, 388–402.

Li, Zhiqiang, Jing, Xiao-Yuan, Zhu, Xiaoke, 2018. Progress on approaches to software defect prediction. Iet Softw. 12 (3), 161–175.

Li, Ming, et al., 2012. Sample-based software defect prediction with active and semi-supervised learning. Autom. Softw. Eng. 19 (2), 201–230.

Li, Ke, et al., 2020. Understanding the automated parameter optimization on transfer learning for cross-project defect prediction: an empirical study. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.

Liang, Hongliang, et al., 2019. Seml: A semantic LSTM model for software defect prediction. IEEE Access 7, 83812–83824.

Liu, Mingxia, Miao, Linsong, Zhang, Daoqiang, 2014b. Two-stage cost-sensitive learning for software defect prediction. IEEE Trans. Reliab. 63 (2), 676–686.

Lu, Jie, et al., 2015. Transfer learning using computational intelligence: A survey. Knowl.-Based Syst. 80, 14–23.

Ma, Ying, Qin, Ke, Zhu, Shunzhi, 2014. Discrimination analysis for predicting defect-prone software modules. J. Appl. Math. 2014.

Ma, Ying, et al., 2012. Transfer learning for cross-company software defect prediction. Inf. Softw. Technol. 54 (3), 248–256.

Manivasagam, G., Gunasundari, R., 2018. An optimized feature selection using fuzzy mutual information based ant colony optimization for software defect prediction. Int. J. Eng. Technol. 7 (1.1), 456–460.

Mousavi, Reza, Eftekhari, Mahdi, Rahdari, Farhad, 2018. Omni-ensemble learning (OEL): utilizing over-bagging, static and dynamic ensemble selection approaches for software defect prediction. Int. J. Artif. Intell. Tools 27 (06), 1850024.

Naseem, Rashid, et al., 2020. Investigating tree family machine learning techniques for a predictive system to unveil software defects. Complexity 2020.

Okutan, Ahmet, Yıldız, Olcay Taner, 2014. Software defect prediction using Bayesian networks. Empir. Softw. Eng. 19 (1), 154–181.

Okutan, Ahmet, Yildiz, Olcay Taner, 2016. A novel kernel to predict software defectiveness. J. Syst. Softw. 119, 109–121.

Pachouly, Jalaj, Ahirrao, Swati, Kotecha, Ketan, 2020. A bibliometric survey on the reliable software delivery using predictive analysis. Libr. Philos. Pract. 1–27.

Pendharkar, Parag C., 2010. Exhaustive and heuristic search approaches for learning a software defect prediction model. Eng. Appl. Artif. Intell. 23 (1), 34–40.

Peng, Yi, et al., 2009. Empirical evaluation of classifiers for software risk management. Int. J. Inf. Technol. Decis. Mak. 8 (04), 749–767.

Phan, Anh Viet, et al., 2018. Dgcnn: A convolutional neural network over large-scale labeled graphs. Neural Netw. 108, 533–543.

Philip, Adithya Abraham, et al., 2019. FastLane: Test minimization for rapidly deployed large-scale online services. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE.

Prasad, M.C., Florence, Lilly, Arya, Arti, 2015. A study on software metrics based software defect prediction using data mining and machine learning techniques. Int. J. Database Theory Appl. 8 (3), 179–190.

Punitha, K., Latha, B., 2016. Sampling imbalance dataset for software defect prediction using hybrid neuro-fuzzy systems with naive Bayes classifier. Teh. Vjesnik 23 (6), 1795–1804.

Qiu, Shaojian, et al., 2019. Transfer convolutional neural network for cross-project defect prediction. Appl. Sci. 9 (13), 2660.

Raghava, Y. Venkata, Rao, Rama Devi Burri, Prasad, V.B.V.N., 2019. Machine learning methods for software defect prediction a revisit.

Rana, Zeeshan Ali, Awais Mian, M., Shamail, Shafay, 2015. Improving recall of software defect prediction models using association mining. Knowl.-Based Syst. 90, 1–13.

Rathore, Santosh S., Kumar, Sandeep, 2019. A study on software fault prediction techniques. Artif. Intell. Rev. 51 (2), 255–327.

Ren, Jinsheng, et al., 2014. On software defect prediction using machine learning. J. Appl. Math. 2014.

Rodriguez, Daniel, et al., 2013. A study of subgroup discovery approaches for defect prediction. Inf. Softw. Technol. 55 (10), 1810–1822.

Saifan, Ahmad A., Abu-wardih, Lina, 2020. Software defect prediction based on feature subset selection and ensemble classification. ECTI Trans. Comput. Inf. Technol. (ECTI-CIT) 14 (2), 213–228.

Shen, Zhidong, Chen, Si, 2020. A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. Secur. Commun. Netw. 2020.

Siers, Michael J., Islam, Md Zahidul, 2015. Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. Inf. Syst. 51, 62–71.

Siers, Michael J., Islam, Md Zahidul, 2018. Novel algorithms for cost-sensitive classification and knowledge discovery in class imbalanced datasets with an application to NASA software defects. Inform. Sci. 459, 53–70.

Siers, Michael J., Islam, Md Zahidul, 2018b. Novel algorithms for cost-sensitive classification and knowledge discovery in class imbalanced datasets with an application to NASA software defects. Inform. Sci. 459, 53–70.

Singh, Malkit, Salaria, Dalwinder Singh, 2013. Software defect prediction tool based on neural network. Int. J. Comput. Appl. 70 (22).

Sobrinho, de Paulo, Vicente, Elder, Lucia, Andrea De, Maia, Marcelo de Almeida, 2018. A systematic literature review on bad smells—5 W's: which, when, what, who, where. IEEE Trans. Softw. Eng..

Sun, Zhongbin, Song, Qinbao, Zhu, Xiaoyan, 2012. Using coding-based ensemble learning to improve software defect prediction. IEEE Trans. Syst. Man Cybern. C 42 (6), 1806–1817.

Tabassum, Sadia, et al., 2020. An investigation of cross-project learning in online just-in-time software defect prediction. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering. ICSE, IEEE.

Taek, Lee., et al., 2016. Developer micro interaction metrics for software defect prediction. IEEE Trans. Softw. Eng. 42 (11), 1015–1035.

Tantithamthavorn, Chakkrit, et al., 2018. The impact of automated parameter optimization on defect prediction models. IEEE Trans. Softw. Eng. 45 (7), 683–711.

Tiwari, Sadhana, Singh, Birmohan, Kaur, Manpreet, 2017. An approach for feature selection using local searching and global optimization techniques. Neural Comput. Appl. 28 (10), 2915–2930.

Vashisht, Vipul, Lal, Manohar, Sureshchandar, G.S., 2016. Defect prediction framework using neural networks for software enhancement projects. J. Adv. Math. Comput. Sci. 1–12.

Wahono, Romi Satria, Herman, Nanna Suryana, 2014. Genetic feature selection for software defect prediction. Adv. Sci. Lett. 20 (1), 239–244.

Wahono, Romi Satria, Herman, Nanna Suryana, Ahmad, Sabrina, 2014. Neural network parameter optimization based on genetic algorithm for software defect prediction. Adv. Sci. Lett. 20 (10–11), 1951–1955.

Wang, Huanjing, Khoshgoftaar, Taghi M., Napolitano, Amri, 2010. A comparative study of ensemble feature selection techniques for software defect prediction. In: 2010 Ninth International Conference on Machine Learning and Applications. IEEE.

Wang, Shuo, Yao, Xin, 2013. Using class imbalance learning for software defect prediction. IEEE Trans. Reliab. 62 (2), 434–443.

Wang, Tiejian, et al., 2016. Multiple kernel ensemble learning for software defect prediction. Autom. Softw. Eng. 23 (4), 569–590.

Wenjie, Li, 2019. Imbalanced data optimization combining K-means and SMOTE. Int. J. Perform. Eng. 15 (8), 2173.

Wu, Yumei, et al., 2020. LIMCR: Less-informative majorities cleaning rule based on Naïve Bayes for imbalance learning in software defect prediction. Appl. Sci. 10 (23), 8324.

Xia, Xin, et al., 2016. Hydra: Massively compositional model for cross-project defect prediction. IEEE Trans. Softw. Eng. 42 (10), 977–998.

Xu, Zhou, et al., 2018a. Cross version defect prediction with representative data via sparse subset selection. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension. ICPC, IEEE.

Xu, Zhou, et al., 2018b. HDA: Cross-project defect prediction via heterogeneous domain adaptation with dictionary learning. IEEE Access 6, 57597–57613.

Xu, Zhou, et al., 2019. Software defect prediction based on kernel PCA and weighted extreme learning machine. Inf. Softw. Technol. 106, 182–200.

Yadav, Harikesh Bahadur, Yadav, Dilip Kumar, 2015. Construction of membership function for software metrics. Procedia Comput. Sci. 46, 933–940.

Yatish, Suraj, et al., 2019. Mining software defects: should we consider affected releases? In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE.

Yousef, Ahmed H., 2015. Extracting software static defect models using data mining. Ain Shams Eng. J. 6 (1), 133–144.

Yu, Qiao, Jiang, Shujuan, Zhang, Yanmei, 2017a. The performance stability of defect prediction models with class imbalance: An empirical study. IEICE Trans. Inf. Syst. 100 (2), 265–272.

Yu, Qiao, et al., 2017b. A feature selection approach based on a similarity measure for software defect prediction. Front. Inf. Technol. Electron. Eng. 18 (11), 1744–1753.

Yu, Tingting, et al., 2018. Conpredictor: Concurrency defect prediction in real-world applications. IEEE Trans. Softw. Eng. 45 (6), 558–575.

Yu, Qiao, et al., 2020. Process metrics for software defect prediction in object-oriented programs. IET Softw. 14 (3), 283–292.

Zhang, Zhi-Wu, Jing, Xiao-Yuan, Wang, Tie-Jian, 2017. Label propagation based semi-supervised learning for software defect prediction. Autom. Softw. Eng. 24 (1), 47–69.

Zhang, Zhi-Wu, Jing, Xiao-Yuan, Wu, Fei, 2018a. Low-rank representation for semi-supervised software defect prediction. IET Softw. 12 (6), 527–535.

Zhang, Hongyu, Nelson, Adam, Menzies, Tim, 2010. On the value of learning from defect dense components for software defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering.

Zhang, Xueying, et al., 2015. A dissimilarity-based imbalance data classification algorithm. Appl. Intell. 42 (3), 544–565.

Zhang, Yun, et al., 2018b. Combined classifier for cross-project defect prediction: an extended empirical study. Front. Comput. Sci. 12 (2), 280.

Zheng, Shang, et al., 2020. Software defect prediction based on fuzzy weighted extreme learning machine with relative density information. Sci. Program. 2020.

Zhou, Yun, Fenton, Norman, Neil, Martin, 2014. BayesIan network approach to multinomial parameter learning using data and expert judgments. Internat. J. Approx. Reason. 55 (5), 1252–1268.

Zhou, Lijuan, et al., 2018b. Imbalanced data processing model for software defect prediction. Wirel. Pers. Commun. 102 (2), 937–950.