IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 20 October 2023, accepted 6 November 2023, date of publication 20 November 2023, date of current version 30 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3335175

RESEARCH ARTICLE

Smell-Aware Bug Classification

KHYBER[1], SIKANDAR ALI[1], FAZLI WAHID[1], SAMAD BASEER[2], AHMED ALKHAYYAT[3], AND AKRAM M. AL-RADAEI[4]

[1]Department of Information Technology, The University of Haripur, Haripur 22620, Pakistan
[2]Department of Computer System Engineering, University of Engineering and Technology, Peshawar 25000, Pakistan
[3]College of Technical Engineering, The Islamic University, Najaf, Iraq
[4]Information Technology Department, Thamar University, Yemen

Corresponding authors: Akram M. Al-Radaei (akram.alradaei@tu.edu.ye) and Sikandar Ali (sikandar@uoh.edu.pk)

ABSTRACT Code smell indicates inadequacies in design and implementation choices. Code smells harm software maintainability including effects on components' bug proneness and code quality has been demonstrated in previous studies. This study aims to investigate the importance of code smell metrics in prediction models for detecting bug-prone code modules. For improvement of the bug prediction model, in this study, smell-based metrics of code have been used. For the training of our model, we employed 14 different open-source projects from the PROMISE repository. Every project file consists of source code as well as smell code metrics and was written in Java. We examined different evaluation metrics such as F1_score, accuracy, precision, recall, the area under the receiver operating characteristic curve, and the area under the precision-recall curve of the five methods within the version, within the project, and across the projects. We classify the code components as buggy or non-buggy using Naïve Bayes, Random Forest (RF), Support Vector Machine (SVM), Logistic Regression, and k-Nearest Neighbor classifiers. RF and SVM have given better results within the version as well as within the project.

INDEX TERMS Code smell, source code, smell-aware, bugs classification.

I. INTRODUCTION

The software system has an essential role in our daily life. Software systems are used to achieve almost all daily requirements. The software system is widely used for different tasks in this digital world, the world is powered by software. Human beings use software systems for economics, transport, medicare, communication, knowledge, combat, power plants, or even for the entertainment of humans. Since human beings depend primarily on software, software applications' accurate functionality is vital, and as far as possible it should be bug free. A bug in software is a flaw or malfunction in a software code that causes incorrect or unwanted results [1]. Software defects are called bugs in a software development process. It is unanticipated deeds, and actions figured out by the quality control engineers in application testing and are preserved as software bugs. Bugs have high effects on software quality [2]. The information related to bugs is kept in a bug report [2], [3]. When the software bug reports are generated, the reports are returned to the software development team to fix the identified bugs. This assignment and transfer procedure is term as bug triaging [4], [5]. The process of bug fixing is exceptionally steady and time-consuming. It is required to detect bugs automatically through automatic bugs detection that is binary classification. The code may belong to the buggy or non-buggy classes in binary classification. It is technically conceivable to create an application without bugs, however, this is not the case in practice [6]. Varshneya [6], proposed that making an application without bugs is impossible because complete code coverage is not a criterion for bug detection unless that software is a life-critical application. Even though that will be impossible to make the application entirely without bugs, software engineers strive to release applications with the fewest possible bugs. Hence, software testing must be an obligatory portion of the software development life cycle (SDLC).

Predicting software bugs is essential in software development because predicting buggy modules before program release increases overall software quality and user satisfaction and further improve the whole software performance [7].

The associate editor coordinating the review of this manuscript and approving it for publication was Alberto Cano.

Furthermore, software adaptation to various surroundings is improved early by predicting software bugs and optimizing resource utilization. Several techniques are proposed to deal with software bug prediction problems. For the prediction of software bugs, machine learning (ML) techniques are well documented [1], [2], [3], [4], [5], [6], [7]. They are widely used to predict buggy components based on historical data, essential metrics, and other software computing techniques. This research paper uses five supervised machine learning classifiers to evaluate ML abilities in software bug prediction.

The primary goal of this investigation is to create a bug prediction model by using code smells as a candidate metric [1]. The bug prediction model proposed in this study is smell aware, i.e., to categorize the code into two classes: buggy class and non-buggy class based on source code and smell code metrics. On the contrary, the smell-aware prediction can decrease the debugging period by localizing and refactoring the smelly files triggering the failure [8]. To be smell-aware, we added an intensity index to the dataset. In this study, both source code and smell code metrics are used to train a bug prediction model. We used Logistic Regression (LR), Naïve Bayesian (NB), Random Forest (RF), and Support Vector Machine (SVM) algorithms as our selected algorithms to train the bug prediction models. Furthermore, in this study, we compare the efficiency of the NB classifier, SVM classifier, LR classifier, and RF classifier. The comparison was made based on specific measures such as accuracy, precision, F1 score, recall, and ROC curves.

A. PROBLEM STATEMENT

Code smell often indicates a more severe problem in the software system. It arises when the software engineer does not follow the design principles, for example, encapsulation, modularity, abstraction, hierarchy (top down and bottom-up strategy), modifiability, cohesion, and coupling. Code smell lowers code quality and makes it difficult to understand and sustain [8]. Code smell exposes design flaws and creates a more challenging software system to understand, maintain, and improve [9]. For this purpose, we want to detect and classify the bugs based on both smells-based and sourced code-based metrics, which is an active research area in software engineering. Some studies on bug classification have been published [10], [11], [12], [13] however, the majority of them are limited to the source code metrics only [14], [15], [16] which lowers the predictive ability of the preceding bug classification model. This study develops a model, which is based on both source and smell code metrics. To be smell aware we added the intensity index. The intensity index estimates the severity of code smell that aids in bug classification and showcases the complexity of the code. To develop a smell aware bug prediction model, the intensity index plays an important role in deciding the severity of design issues influencing a code module. Most of the published literature [10], [13], [16] did not use intensity index as a feature for bug classification, our essential contribution to the dataset is the addition of an intensity index for smell-aware bug classification.

In the published literature majority of the studies use LR [1], [17], NB [13], [16], k-NN [18], and DTrees [16] as bug classification models, however, the effectiveness of SVM was not explored in the published literature. Furthermore, the preceding models were trained on dataset that were based only on source code metrics.

B. AUTOMATED BUG PREDICTION

The first bug prediction model is designed by Taba et al. [19]. They specifically established three measures, which they coined as antipattern metrics. These metrics are described in the context of smells and might be used to assess the average amount of antipatterns, complications, and repetition length using as antipattern measurements in addition to structural metrics [20]. A bug prediction model will now take advantage of antipattern measures to develop a smell aware bug prediction model. Furthermore, structural metrics were developed and tested with structural metrics, demonstrating that when the design faults are considered, bug prediction models can improve by up to 12.5 %.

We assumed that in a bug prediction model, the severity of a design issue disrupts a source code segment. We employed the intensity index, which was determined by Fontana et al. [21], to prove this conjecture [21]. To create a smell aware bug prediction model, we consider the design flaws and its severity that affect a code module. More precisely, we assessed the severity index's predictive power by combining it with a bug prediction model based on structural quality measures [22] and comparing the accuracy with that obtained by the standard model on fourteen large open-source Java projects. The benefits of adding the severity index to these models to other structural metrics, and the ones used to calculate the intensity were also analyzed. According to the findings, using the intensity index to predict the bug improves the classification results. The consequences exposed that based on architectural quality criteria (AQC) using the severity index as a predictor of buggy modules improves the correctness of a bug prediction model. Furthermore, the data show that the severity index is more significant than every other quality metric in predicting the bug-proneness of the smelly classes. The findings show that using the severity index as a reliable indicator of buggy modules improves the effectiveness of structurally based baseline bug prediction models. Still, they emphasize the significance of the intensity of code smells in the process metrics-based prediction approaches.

C. OBJECTIVE OF THE STUDY

- The first objective of this research work is to leverage code smell metrics as the feature metrics for the development of a bug prediction model specifically called smell aware bug prediction model. Therefore, the model in this study is developed using various sources of information, specifically the product and process metrics.

- The second main objective of the study is to describe the real contribution made by the intensity index to the bug classification. The intensity index estimates the severity of code smell that aids in bugs classification and shows the complexity of the codes. To develop a model for bug classification, the intensity index plays an important role in deciding the severity of design issues influencing a module. The experiments conducted in this study aims to see how vital the intensity index is in prediction models for detecting bug-prone code modules [20].
- The third objective of the study is to develop a smell aware bugs prediction model by highlighting and evaluating the selected models based on the modified data set based on smell-based metrics such as intensity index and amongst them specify the best model. Therefore, the smell-aware bug classification model was built using candidate classifiers such as NB, RF, SVM, k-NN, and LR classifiers. For this objective, this study examines different evaluation metrics of the five models within the version, within the project, and across the projects.

II. BACKGROUND AND LITERATURE REVIEW

A. CODE SMELL

The term 'Code Smell' was originally coined by Fowler in his refactoring book. Code smells are a metaphor for defining patterns, commonly linked with poor design and bad programming practices [23]. Code smell often indicates a more severe problem in the system. It arises when the software engineer does not follow the design principles, for example, encapsulation, modularity, abstraction, hierarchy (top down and bottom-up strategy), modifiability, cohesion, and coupling. Even though the developers know the design principles, the developers often violate them because software engineers do not have experience, the pressure of a deadline, and heavy competition between competitors in the marketplace. In the real world, software systems regress daily to meet new requirements or correct bugs discovered. The pressure to fulfill tight deadlines makes it difficult for developers to manage the complexity of such modifications effectively. Indeed, development operations are carried out frequently in an undisciplined manner, resulting in eroding the system's initial design by presenting technical debts [24]. Software aging is a common term for this phenomenon [25]. This phenomenon was measured in terms of entropy by some researchers. Fowler et al. [26] defined this phenomenon as ''Bad code smell'' (shortened as ''smells of code'' or simple ''smell'') as ''signs of existence of the bad design or choices of implementation applied in a software application development''.

Smell means developers/designers do not appropriately design the software. Code smells have different kinds, such as long methods, complex classes, message chains, and many more, which are explained in the subsequent sections. These are just a few instances of code smells that might harm a software system. In addition to this approach for automatically detecting code smells in source code [27], the community of researchers struggled to understand code smell and its adverse effects on the non-functional characteristics of source code. We can learn when and why code smells take shape, how they develop and persist in software programs, and to what extent code smells apply to software developers. Several studies [19], [20], [21] have also reported that code smells can have adverse effects on software maintainability and understandability. Khomh et al. [28] and Palomba et al. [29] recently recognized that classes with design flaws are more likely to compromise future bugs. Moreover, this research revealed the hidden, and underhanded impact of code smells on bug prediction. The academic community has just scratched the surface of these observations.

B. MOTIVATION AND NOVELTY

In real-life, software applications frequently change to be adapted to novel requirements or as a result of bug fixes. The demand to fulfill tight deadlines makes it difficult for programmers to effectively deal with the complexities of such modifications. Indeed, development processes are carried out in an undisciplined manner, resulting in eroding the system's preliminary design by introducing technical debts [24]. It is experimentally proven that code smells cause code to be less understandable. The empirical evidence shows that code smell has been exposed to hamper code understandability [30], raise modification [28], and proneness of error [31], and make code less maintainable [32]. Code smell impacts normal software development tasks such as code inspection, refactoring, and maintenance [33]. Some researchers [15], [20], [21] have termed this issue as code smell.

With code smell, the system may work, but it might slow down the entire system and can produce future bugs due to bad design and smell. When the bugs grow, the system will get an error and give an unwanted result. Code smells are signs to identify poor designs that result in having code with a smaller amount of maintainability. There is huge possibility that something may be assumed in the source code without following the actual design pattern when we have more signs of bad code smells in the source code. Therefore, we want to develop a smell-aware bug prediction. Most of the authors did bug classification without smell based metrices i.e., some of them did bug classification based on priority and some of them did bug classification based on severity and some others did it using different methods and approaches. In this study, we develop a smell-aware bug classification using different supervised ML classifiers.

C. LITERATURE REVIEW

One of the hottest current research fields in software engineering is bug prediction. The academic community has created a variety of prediction methods. The major approaches to software bug prediction are based on classification. It was thought that the software's complexity might lead to defects. To highlight the complexity of software, Akiyama

[34] suggested a basic model based on LOC. It was too simplistic to use LOC as a bug prediction metric. In 1976, MaCabe proposed cyclamate complexity (CC) metrics for bug prediction [35]. At that time, Halstead and CC [36] were outstanding measurements; unfortunately, they suffer from severe flaws. Khoshgoftaar and Munson et al. [37] suggested a more accurate categorization model. In the 2000s, process metrics prediction models were introduced as the use of version control systems expanded. The bug prediction model created in the 2000s had several disadvantages. One shortcoming of this model is that it cannot predict defects when a source code file is modified. To solve this issue, the Just in Time (JIT) model is proposed to predict bugs. Another disadvantage was anticipation of bugs for new projects or projects with limited historical data. As a solution to this constraint, cross-defect prediction methods were developed. This approach demonstrates that when cross-company data increases the likelihood the percentage of false positives also increases.

Pan et al. [14] proposed 13 program-slicing metrics for bug classification in the C programming language; these metrics use program slice information to count program size, coupling, sophistication, and cohesion. Program slicing metrics have measurements for program behavior in contrast to standard code metrics that focus on statements of code or structure of code. The program slicing techniques [38], [39] investigate the behavior of source code by looking at the flow and control dependencies between statements. Some metrics used in program slicing lists as sliceCount, verticesCount, edgesCount, sliceVerticesSum, globalInput, lackOfCohesion directFanIn, and edgesToVerticesRatio. Program-slicing metrics measurements have an overall accuracy of 82.6 percent for the Apache HTTP project and 92 percent for the Latex2rtf project at the file level, respectively. One of the significant drawbacks of program-slicing metrics metrics is that they can only use them to generate preprogram-slicing data for large projects. Regarding bug classification, the data imply that program-slicing metrics measurements are at least as effective as UC metrics.

Bug fixing is a time-consuming process. The bugs must be grouped into several categories to make this procedure easier [10]. Binary categorization is one of the most fundamental software bug classifications, wherein a software code is labeled either as a buggy or clean code. To identify software codes as buggy or non-buggy, the proposed approaches use machine learning algorithms, discriminative words, and a fuzzy similarity metric with a user-defined threshold value. The researcher applied various techniques with dissimilar parameters over the Kaggle dataset. SLC classifiers outperform other classifiers in all aspects.

At the primary stage of app development, the software bug prediction model advances the essential parts, for example, reliability, software quality, and efficiency, and reduces the development cost [16]. Bugs constitute a crucial barrier to system consistency and efficiency in most software systems, which become increasingly vast and sophisticated programs. The classifiers, LR, NB, and Decision Tree are used to construct a model to predict the occurrence of software bugs based on the historical data using four supervised machine learning algorithms. Among the many software metrics presented are Metrics of Dimension, Metrics of Complexity, Metrics of Object-Oriented, and Metrics of Android-oriented. Dimensional metrics make available quantitative metrics linked with software sizes like code size and modularity [16]. The number of Byte-code Instructions (NBI), Number of Classes (NOC), Number of Methods (NOM), and Instructions per Method (IPM) are the metrics used in this category's analysis. They gathered information from projects available on the GitHub platform. The accuracy of distinct classifier models is lower. Across multiple samples, the models are not tested. As a result, taking random samples is likely to get lower accuracy. They utilized four algorithms, with the random forest providing the best results.

At the primary stage of app development, the software bug prediction model advances the essential parts, for example, reliability, software quality, and efficiency, and reduces the development cost [16]. Bugs constitute a crucial barrier to system consistency and efficiency in most software systems, which have become increasingly vast and sophisticated programs. The classifiers LR, NB, and Decision Tree are used to construct a model to predict the occurrence of software bugs based on the historical data using four supervised machine learning algorithms. Among the many software metrics presented are Metrics of Dimension, Metrics of Complexity, Metrics of Object-Oriented, and Metrics of Android-oriented. Dimensional metrics make available quantitative metrics linked with software sizes like code size and modularity [16]. Several Byte-code Instructions (NBI), Number of Classes (NOC), Number of Methods (NOM), and Instructions per Method (IPM) are the metrics used in this category's analysis. They gathered information from projects available on the GitHub platform. The accuracy of distinct classifier models is lower. Across multiple samples, the models are not tested. As a result, taking random samples is likely to get lower accuracy. They utilized four algorithms, with the random forest providing the best results.

Classes involving smells are revised more commonly than any other classes, according to Khomh et al. [40]. According to Olbrich et al. [41], Smelly-code components need more attention and have different alteration behavior. Smells can be regularly detected using automated technologies. Smells can also be identified and analyzed in massive code bases using tools. As a result, a diversity of closed-source and open-source smell detection technologies have been established. Even though there are various tools available nowadays, each tool captures only a subset of smells. No tool is pre-programmed to do identification of entirely smells [42]. The smells detected by the tools have a slight overlap. No single tool can detect all of the smells we investigated. It's impossible to tell which detection method is optimal for

real-world systems. Hall et al. [42] studied the carry out of 5 smells on the number of faults in three systems, and that the only conclusion that was steady transversely entirely three systems would be that Switch Statements had no bearing on problems from at all of the systems.

Khomh et al. [28] also discovered that classes with design flaws (''antipatterns'') are much more likely to include bugs in the future. Though this research displayed the efficacy of code smells in bug detection, the discoveries have still to be incorporated into bug prediction models. The research authors, Palomba et al. [22], evaluated the involvement of a metric of the severity of code smells by adding it to current bug prediction model and comparing the findings of the novel model to the baseline model. In this paper, ML classification methods predicted two types of bugs: buggy and non-buggy. Multilayer Perceptron, ADTree, Naïve Bayes (NB), LR algorithm, Decision Table Majority, and Simple Logical were some of the classifiers they investigated. They employed the intensity index, which is identified by Fontana et al. [21]. The index is calculated by JCodeOdor, a code smell detector that uses detection techniques applied to metrics. JCodeOdor produces five expressive values to be used per threshold values: VERY-LOW, LOW, MEAN, HIGH, and VERY-HIGH. In this paper, the author added an intensity index with structural metrics of the source code. The intensity index's contributions to bug estimation techniques are based on process metrics. In bug prediction models built on product metrics, process metrics, or a grouping of both, the intensity index assists in distinguishing bug-prone code components influenced by code smells.

Reference [23] defines technical debt as a circumstance where software engineers accept giving up one dimension of a software product (namely, quality) to maximize another (i.e., applying a group of novel attributes before a time limit). Even if this sacrifice brings immediate rewards, the debt must eventually be paid off. When there is too much technical debt, it slows down development and makes code more difficult to maintain. One type of technical debt is code smells. In this paper, Ubayawardana and Karunaratn [1] used several metrics of source code and metrics of code smell-based to construct a bug prediction model. They trained the model on different versions of 13 different Java programming language open-source projects utilizing NB, LR classifier, and RF approach as viable techniques. They demonstrated that when paired with source code metrics, smelly code metrics can pointedly advance the accuracy of the bug prediction model. The RF algorithm-based model outperformed compared to other algorithms in terms of precision and accuracy within a version, within a project, and across the projects. They employed two metrics for bug prediction, one for code and the other for the process. Process metrics gather information from VCSs like GitHub and issue-tracking systems like Bugzilla, whereas code metrics are obtained from source code. To improve traditional bug prediction methods, they incorporated smell-based measures.

III. PROPOSED METHODOLOGY

The research community has presented many bugs prediction [1], [2], [3], [4] and classification [5], [6] models based on various indicators to recognize more error-prone modules in software applications. Few of them [7] have enhanced accuracy and evaluation metrics as compared to others. However, only a few authors [8] did bug classification but their model is not smell-aware. This study used different approaches to do smell-aware bug classification through ML algorithms. Furthermore, we will do the result analysis of the algorithms with each other and compare their accuracy using dissimilar source code and smell-based metrics.

We propose five ML models: LR, RF, SVM, NB, and k-NN, to detect and classify smell-aware bugs. Our objective in these proposed models is to achieve high accuracy. Multiple stages have been conducted to address the challenges in Machine Learning, resulting in significant success in achieving the highest possible accuracy for smell-aware bug classification. However, we aim to investigate the reasons behind the lower accuracy of the ML models and compare the results and performance of LR, SVM, RF, k-NN, and NB. Consequently, we will analyze which machine learning approach is best for smell-aware bug detection and classification.

For our study, we proceeded with the dataset from Jureczko et al. [43], which is accessible from the PROMISE repository [44]. This dataset comprises a rich collection of 44 releases from 14 projects, each with 20 code metrics. Additionally, the occurrence of bugs in each release is readily available. It is worth noting that the dataset includes systems of various sizes and scopes, allowing us to enhance the validity of our investigation [45]. Furthermore, we considered the findings of Mende et al. [46], who discovered that models trained on limited datasets can yield unreliable performance estimations.

In this study, we utilize source code metrics to develop the first smell-aware bug prediction model. To train our initial model, we incorporate various source code metrics discussed in Section II, which is the literature review. The primary objective of this study is to correlate the code smell metrics proposed by [19] and [22]. By associating these metrics, we aim to enhance the predictive power of our improved smell-aware bug prediction model. For bug prediction, we employ five classification models: RF classifier, LR classifier, SVM classifier, NB classifier, and k-NN classifier. Through an extensive evaluation, we demonstrate the effectiveness of the metrics proposed by [19] and [22] in enhancing the predictive power of our developed model.

The proposed methodology for our models consists of several stages, as depicted in Figure 1. The process begins with the input of the dataset, which is then processed by the proposed machine learning techniques. These techniques analyze the dataset and generate output by classifying the code as either buggy or non-buggy. Once the classification is complete, the next stage involves evaluating the performance

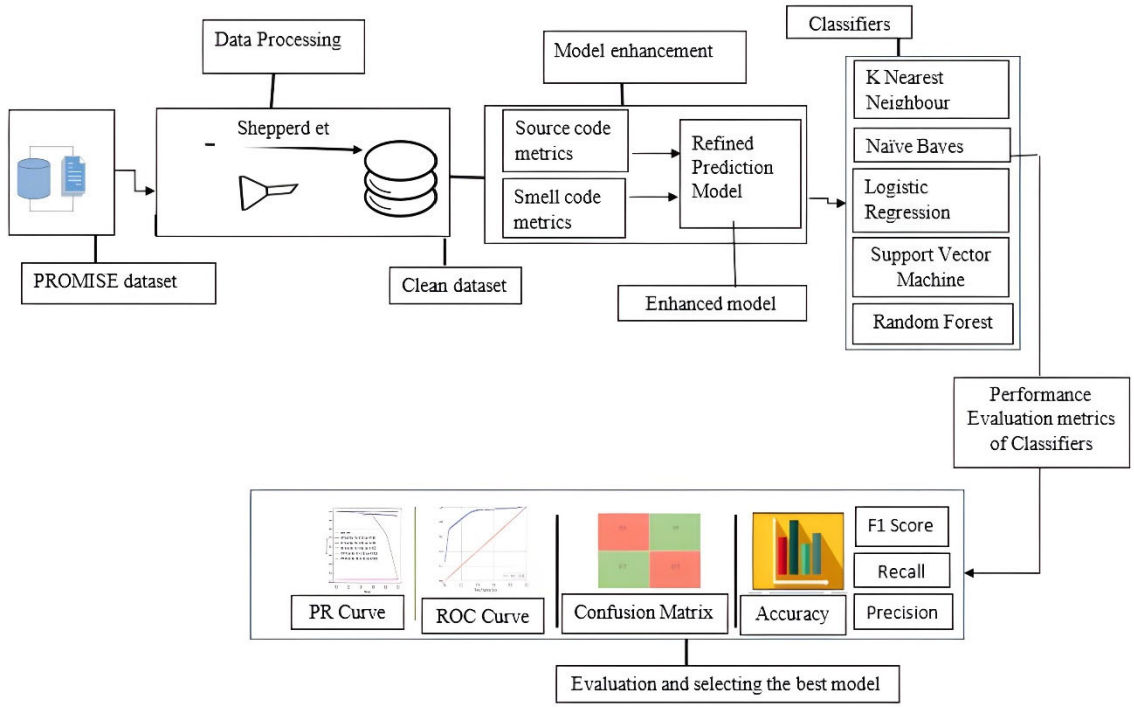


FIGURE 1. Proposed model methodology.

of the proposed models. This evaluation includes comparing the results of our models with those of other existing models. By conducting this comparison, we can assess the effectiveness and efficiency of our proposed models in bug detection and classification. In summary, the proposed methodology involves inputting the dataset, applying machine learning techniques to classify the code, and subsequently evaluating and comparing the performance of our models with other approaches.

A. INPUT

The first step in our methodology is the input stage, where we gather data from different open-source projects containing bugs. These projects are obtained from the PROMISE bug repository and serve as the training data for our model. The details of the software projects dataset used in this study can be found in 3.3. A comprehensive description of the dataset is provided, including specific information about each project. For further reference, please consult Table 1, which presents the specific details of the dataset used in our study.

B. PROPOSED MACHINE LEARNING TECHNIQUES

In this study, we developed a smell-aware bug prediction model using various machine learning approaches. Our chosen learning style is supervised learning, which means we focus on algorithms that support this type of learning. The prediction outputs of our model are classified into two types: classification and regression. Classification involves linking input variables to discrete output values, while regression predictive analysis maps input factors to continuous output variables. In the case of our bug prediction model, the output type is binary, meaning we categorize a source code segment as either buggy or non-buggy. Consequently, we will only explore methods that support binary classification, as this paper specifically focuses on the binary classification of bugs. For our investigation, we selected five commonly used classifiers in bug prediction research: LR classifier, RF classifier, SVM classifier, k-NN, and NB classifier. These classifiers will be utilized in our study to develop and evaluate the performance of the smell-aware bug prediction model.

1) LOGISTIC REGRESSION

The LR (Logistic Regression) algorithm is widely used in data mining, particularly for binary classification tasks. It is a statistical and data mining method that is commonly employed by statisticians and academic researchers to analyze and classify binary and proportional response datasets. Logistic regression is known for its ability to model the relationship between a set of input variables and a binary outcome. Researchers in various fields, such as statistics and data mining, have extensively utilized the LR algorithm to analyze and classify binary data. This technique has proven to be effective in a wide range of applications and is often chosen as a go-to method for binary classification tasks. Studies referenced as [47] and [48] provide further insights into the usage and application of logistic regression in statistical analysis and binary classification.

Input: *Promise repository bug prediction datasets for to traning*

1. *For* $i \leftarrow 1$ *to* k
2. *For every instance of training data* d_i
3. *For the regression set the target value to*
$$Zi \leftarrow \frac{y_j - P(1|d_j)}{\left[P(1|d_j).\left(1 - P(1|d_j)\right)\right]}$$
4. *Initialize instance weight to* $P(1|d_j).(1 - P).P(1|d_j)$
5. *Finalize af*(j) *to the data with class value* (Z_j) *& weights* (w_j)

Classificaiton Label Determination

6. *Assign* $(buggy)$ *if* $P(1|d_j) > 0.5, otherwise\ (non - buggy)$

FIGURE 2. Pseudocode for logistic regression.

LR classifier has several key advantages, including the ability to prepare probabilities and can be extended to handle multi-class classification issues [49], [50]. Another advantage is that most LR model analysis methods are based on the same principles as linear regression [51]. The LR algorithm is a widely used supervised ML classification technique. It works on categorical dependent variables, yielding two discrete variables (0 or 1). As a cost function, the sigmoid function is used. The sigmoid function converts a predicted actual value into a probability value (0-(1).

Logistic Sigmoid function:

$$P(x) = \frac{1}{1 + e^{(-x)}} \tag{1}$$

P(x) is a probability prediction function with a value between 0 and 1, x is the probability function's input (the algorithm's prediction value), and e is Euler's number, which has a value of about 2.71828 as indicated in equation 1.

To predict bugs, a logistic regression (LR) machine learning model is utilized. Initially, the LR model is trained using data from fourteen open-source projects. Subsequently, the model is evaluated with test data to determine its behavior and achieve the highest possible accuracy. The LR model aims to classify the presence or absence of bugs in an application, assigning a category of 1 for true (buggy) and 0 for false (non-buggy). The pseudocode in Figure 2, describes the Logistic Regression which is used to train and test the bug prediction model.

2) THE RANDOM FOREST ALGORITHM

Random forest is an ensembled learning technique that is encircled of n collections of independent decision trees [49]. Traditional machine learning techniques typically result in low classification accuracy and are prone to overfitting. Many people study the algorithm for merging classifiers to enhance accuracy. Many researchers begin their research to improve classification accuracy by merging classifiers. Random Forest is an innovative technique and a new combinational algorithm that is coupled with a succession of tree classifiers, where every tree cast a unit vote for the further most common class which means voting by the majority, and then the findings are merged to achieve the final sorted result [52]. Random Forest has a lot of interesting characters. RF has never been over fitted, has good classification accuracy, and is immune to outliers and noise [52]. Random Forest is widely used for classification and prediction, as well as regression and our main purpose is to use the RF algorithm for binary classification of bugs classification. For classification, the RF algorithm finding is based on the class's mode. In comparison to typical algorithms, Random Forest has several advantages over traditional algorithms.

As a result, Random Forest can be used in a variety of situations. For classification in the terminal leaf nodes or decision nodes when constructing a prediction, the RF algorithm uses multiple trees to calculate the majority votes. Decision trees are essentially tree-like structures; the top node is called the root of the tree, which recursively split at the decision node series from the root until the decision node is reached [53]. The decision tree algorithm divides the dataset into smaller subsets using a top-down, "greedy," methodology. Entropy is calculated to determine which attribute to split on at each node. A tree-like learning method has the benefit of permitting the training of models on large datasets and moreover on both quantitative and qualitative input variables. Furthermore, tree-based models may be resistant to redundant variables or variables with significant correlations that could cause overfitting in other learning algorithms [53]. Bagging is the process of randomly selecting samples with replacement, and it produces a new tree for training. The variance will be reduced and a smoother decision boundary will be created by averaging the findings from the 'n' number of trees [49]. For example, while using the random forest for smell-aware bug classification, every tree will give an estimation of the class label likelihood that it belongs to a particular class (buggy and clean code).

The likelihood will then be averaged over the 'n' trees, and the tree with the highest likelihood will produce the estimated class label (Figure 3) and the RF algorithm produces the buggy instance of the code. To decrease the variance further in the decision boundary the tree should be entirely uncorrelated. The implementation of the RF algorithm in Figure 4 includes the pseudocode for RF formation. As well in Figure 5 include the pseudocode for RF prediction.

3) SUPPORT VECTOR MACHINE

The newest supervised machine learning technique is Support Vector Machine [54]. Reference [55] presents an excellent overview of SVMs, and [56] is a more recent book on SVM. As a result, the SVM classifier is a new way to classify and predict data. Vapnik and Cortes [57] developed this highly popular and powerful classification system. SVMs are identified as maximum margin classifiers so the SVMs find the best segregating hyperplane between two classes (see Fig. 5). So, our problem is also binary classification. The PROMISE bug

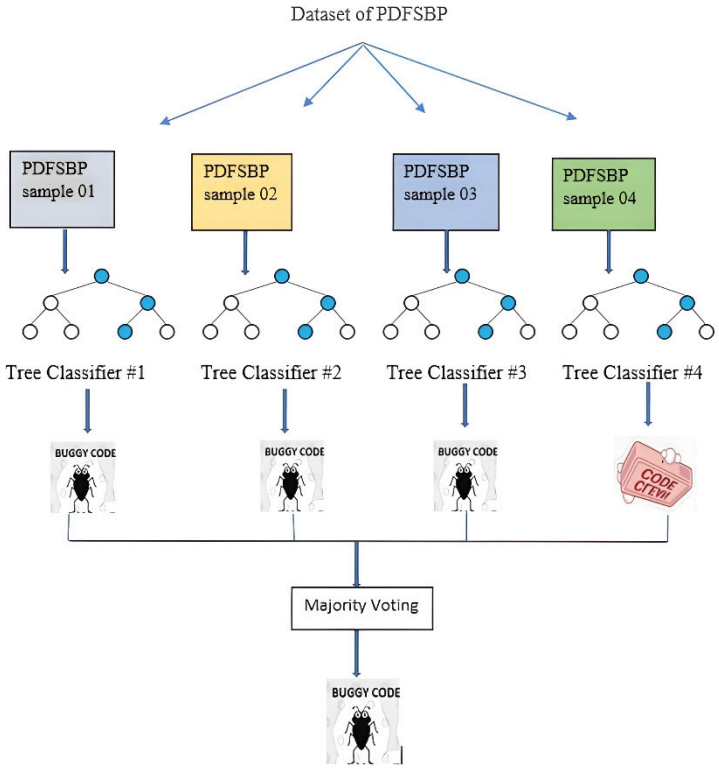


FIGURE 3. ADTrees ensemble technique for bug classification in RF.

Input : *T raining set of PDFSBP with F features*

1. *Randomely pic 'P' features from F features*, $\forall P < F$
2. *By the best split, u sin g ' P 'features, find the 'd' node*
3. *By appling the best split a lg o rithm, break the node int o child nodes*
4. *Iterate steps* : 1 *to* 3 *until* '1' *number of nodes has been reached*
5. *Re a pet the steps* : 1 *to* 4 *and make the forest by of, of generating* 'n' *number of DTrees*
6. *Output* : *R andom Forest Trees* (*RFTs*)

FIGURE 4. Pseudocode for random forest formation.

Input : *P DFSBPfortesting*

1. *Pr e dict & store the findings of each randomly created DTree's on given dataset*
2. *calculate the entirly votes for binary classes* $(buggy, non - buggy)$
3. *Declare majority classas the final result class*
4. *Output*: *final predicted class will be bug or* $non - bug$

FIGURE 5. Random forest prediction pseudocode.

repository dataset is used for bug detection. We will simply cover the method's basic principle in the context of classification using supervised learning techniques. Here, we will merely go over the method's fundamental concept concerning classification using supervised learning approaches. To know the nature of the SVM classifier, one needs to comprehend four main ideas: separating hyperplane, maximum-margin hyperplane, soft margin, and kernel function [58]. When we have a large-scale dataset, it doesn't perform as well because the training time is longer. SVM analysis is divided into three phases:(i) feature selection, (ii) classifier training and testing, and (iii) performance evaluation. It should be noted that these stages are available in most machine-learning approaches and are not exclusive to SVM. For both linear and nonlinear datasets SVM works well. The SVM classifier performs well when the dataset has a huge number of attributes.

SVM works on the fundamental rule of ''margin'', in a nutshell. A distribution between 2 data labels that exist on either side of the hyperplane is built by a hyperplane. The aim is to increase the margins so building enough probable gaps amongst the instances and segregating the hyperplane on both sides of it [59].

Figure 6 segregates dots from triangles, the solid line demonstrates the hyperplane, and the dotted lines running parallel to the solid line demonstrate how far the decision hyperplane can be moved without causing misclassification.

$(W.X + b = 0)$ is a math expression that is a delegation of separating hyperplanes Where, $W = \{w_1, w_2, w_n\}$, symbolized as the weight vector, 'n': number of features; and 'b' stands for a scalar (also referred to as a bias).

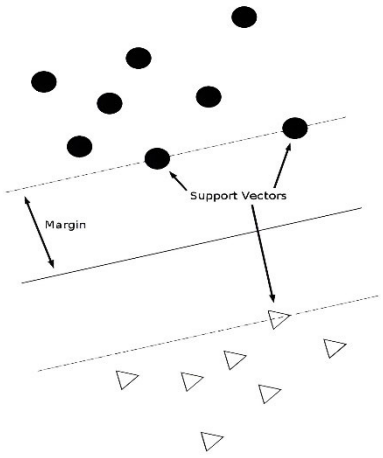


FIGURE 6. Support vectors.

Input : *P DFSBPdataset*
1.Create an SVM-based prediction model for software bugs
2.To get the best SVM parameters, use the optimization
3. *Retreat to optimised SVM model to predict the dataset. If the result of the prediction model hold out to the final condition, stop cylce and give the output of the result.*
Otherwise, go back to step 3

FIGURE 7. Pseudocode for the SVM classifier.

Linearly detachable data: In the condition of linearly detachable data there occurs a pair (W, b) such that.

(i) [W.Xi + b ≥ 1] for yi = 1(Label: class 1)
(ii) [W.Xi + b ≤ -1] for yi = - 1(Label: class - 1)
(iii) MMH (maximum margin hyperplane) can again 28 be written as the decision edge [60], [61].

$$d\left(X^T\right) = \sum_{i=1}^{l} y_i \alpha_i X_i X^T + b0 \tag{2}$$

Representations:

(a) y_i: Xi support Vector class label
(b) X^T is a test tuple
(c) b0 and α_i: numeric parameters
(d) l: number of support vectors [62].

For the SVM classifier in this study, the basic steps are specified in figure 7.

4) NAÏVE BAYES

The Naïve Bayes classification algorithm uses the Bayesian theorem, which is favored when dealing with high-dimensional inputs. We use the Naïve Bayes classification algorithm R function's implementation. For each characteristic X (x1, x2, and x3…xn) the likelihoods are computed by the Naïve Bayes Classifier. Then, as a result, it chooses the instance with the highest likelihood value [63]. For defect prediction the Naïve Bayes are effectively applied in some research efforts. And in this study, it will be applied to software bug prediction as well. The NB approach in machine learning is particularly efficient. The bug prediction binary classification is treated by the NB model, by examining software modules' historical data it trains and constructs the predictor. The predictor is then used to determine whether a new module contains bugs or not. Equation 3 is the Bayes Theorem, and this is derived from conditional probability. The PROMISE bug repository dataset is used in this study, and it is used for binary classification purposes (as buggy and non-buggy data). The dataset has multiple independent features (in our dataset it is called source code and smell code metrics) for example $X = \{x_1, x_2, x_3, \ldots x_n\}$ where x_1 is feature one, x_2 is feature two, and so on. And one dependent feature $Y = \{0, 1\}$, '1' means true = buggy, and '0'means false = non-buggy code. So, the Bayes Theorem will be changed to a binary classification problem.

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)} \tag{3}$$

P (A): Probability of A
P (B): Probability of B
P (A|B): Probability of A when B is given
P (B|A): Probability of B when A is given

Equation 3 we have to change based on our dataset. We will give all input features ($X = \{x_1, x_2, x_3, \ldots x_n\}$) and predict the dependent feature 'y' and categorize it, whether it is buggy or not. Equation 3 can be written as

$$P(Y \mid X) = \frac{P(X \mid Y)\, P(Y)}{P(X)} \tag{4}$$

As we know, $X = \{x_1, x_2, x_3, \ldots x_n\}$. Equation 5 can be reaped from equation 4.

$$\begin{aligned} &the\ P(y \mid x_1, x_2, x_3, \ldots x_n) \\ &\quad = \frac{P(x_1 \mid y)\, P(x_2 \mid y) \ldots\ldots P(x_n \mid y) * P(y)}{P(x_1)\, P(x_2) \ldots . P(x_n)} \\ &P(y \mid x_1, x_2, x_3, \ldots x_n) \\ &\quad = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1)\, P(x_2) \ldots P(x_n)} \end{aligned} \tag{5}$$

The $P(x_1)\, P(x_2) \ldots P(x_n)$ can be considered as constant because this will be the same for every record. So, $P(x_1)\, P(x_2) \ldots P(x_n)$ will be directly proportional to the $P(y) \prod_{i=1}^{n} P(x_i \mid y)$. In order to find out the output of $X = \{x_1, x_2, x_3, \ldots x_n\}$ particular values, we need to take the argmax of $P(y) \prod_{i=1}^{n} P(x_i \mid y)$. Finally, we achieved equation 6 for NB classifier.

$$y = \arg max((y) \prod_{i=1}^{n} P(x_i|y)) \tag{6}$$

Argmax means which will be given the maximum likelihood to consider that. Suppose for True it is given as '0.7' and for False it is given as '0.3', now in this case I will consider '0.7', so the output for the $X = \{x_1, x_2, x_3, \ldots x_n\}$ this particular feature will be Tr.

5) k-NEAREST NEIGHBOUR (k-NN)

The idea behind Nearest Neighbor Classification is simple. According to the class of their closest neighbors, instances are classified. Because that is typically convenient to consider further than one neighbor, the technique is more commonly known as k-Nearest Neighbor (k-NN) Classification, in which k nearest neighbors are applied to determine the class [64]. The algorithm needs the training samples at runtime so they must be in memory at runtime. It is called a lazy learning approach as well. The main points of the k-NN classifier are a smaller amount of computation time and effortlessness of interpretation for the training of model but in the testing phase, it will take a longer time. The value of K is significant in the k-NN algorithm and is used to finetune the algorithm. When the value of K reduces, the model becomes less consistent; conversely, when the value of K grows, the model becomes more stable [65]. When the number of samples or examples rises, the k-NN algorithm becomes slower. To determine the distance among classes, the k-NN method employs the Euclidean distance formula.

C. DATASET DESCRIPTION AND PRE-PROCESSING

Pre-processing of data is a crucial step in the data mining process as training datasets often contain imperfections such as faults, outliers, missing data, and noise. Tools are necessary for detecting and correcting these issues. Raw and unprocessed datasets are typically inadequate and may contain errors, missing data, outliers, and additional noise. To address these concerns, it is essential to evaluate the quality and accuracy of the data before conducting experiments. Pre-processing operations encompass various tasks, including data clean-up, data integration, data transformation, and data reduction. These operations aim to improve the overall quality and reliability of the data, ensuring that subsequent analysis and modeling steps are based on accurate and consistent data. Pre-processing operations include data clean-up, data integration, data alteration, and data lessening [66].

Proper data preparation is needed to advance the accuracy of the training model. Each file has an identical set of properties and is in Comma Separated Values (CSV) format. The attributes in a PDFSBP(PROMISE dataset for software bugs prediction) are as follows: All source code metrics can be obtained in a data file, including project name, version, file name, bug count per file, is Buggy File, WMC, DIT, NOC, CBO, RFC, DAM, MOA, MFA, LCOM, Ca, Ce, NPM, LOC, DAM, MOA, MFA, CAM, ACC. ANA, ACM, ARL, and antipattern cumulative pairwise diversity are the code smell-based metrics provided in a data file (ACPD). In addition to the metric information, each dataset has some metadata. The bug prediction model is unaffected by some of the attributes seen in each file.

For this study, we used data from publicly accessible data repositories. As a result, the data sets have been stripped of project, file, and version names. A file is deemed an 'isBuggyFile' in the data set if at least one problem has been reported against it in a certain version. Table 1 shows the datasets and versions of several projects that we collected for the bugs classification.

TABLE 1. PROMISE bug repository dataset.

Project name	Version Numbers
Apache Ant	1.3, 1.4, 1.5, 1.6, 1.7
Apache Ivy	2
Apache Camel	1.0, 1.2, 1.4, 1.6
Apache Log4j	1.0, 1.1, 1.2
Apache Forrest	0.7, 0.8
Apache POI	1.5, 2.0, 2.5.1, 3.0
Apache Lucene	2.0, 2.2, 2.4
Apache Synapse	1.0, 1.1, 1.2
Apache Velocity	1.4, 1.5, 1.6.1
Apache Tomcat	6
Apache Xalan	2.4, 2.5, 2.6, 2.7
jEdit	3.2, 4.0, 4.1, 4.2, 4.3
Apache Xerces	1.2, 1.3, 1.4.4
Apache pbeans	1.0,2.0

FIGURE 8. Data set division illustration diagram.

In a subsequent version, the same file could be a non-buggy file. The number of problems that have been reported against a file has also been kept track of.

The 'Bugs count per file' attribute was also removed because predicting the number of bugs associated with a file was outward to the scope of this study. To increase accuracy, the data must be in numerical format. As a result, we focused solely on numerical properties when developing the model. Some of the characteristics have stronger correlations with one another. Therefore, those attributes have a high correlation with one another, we can drop one of them.

D. DATA SPECIFICATION

The purpose of this exploration is to look at the model within a version, even within a project and across the projects. Each dataset was broken down into two parts. The model was trained using 70% of each dataset, then tested with 30% of each dataset. The dataset division illustration is in Figure 8.

For instance, Inside the Apache Ant 1.7 version 745 instances were using real data, we first eliminated 5 occurrences of this dataset at random. The accuracy of the prediction was evaluated with a real dataset. 70 percent of instances (518 instances) were used to train the model and

TABLE 2. Source code and smell code-based metrics.

Source Code Metrics	Code Smell Based Metrics
Depth of Inheritance Tree	Average Number of Antipatterns
Weighted Method Count	Intensity Index
Number of Children	Average Number of Antipatterns
Response for a Class	Antipattern Complexity Metric
Coupling Between object classes	Antipattern Recurrence Length
Lack of Cohesion of Methods (LCOM)	Antipattern Cumulative Pairwise Differences
Lack of Cohesion of Methods (LCOM3)	
Number of Public Methods	
Data Access Metric	
Measure of Aggregation	
The measure of Functional Abstraction	
Cohesion Among Class Methods	
Inheritance Coupling	
Coupling Between Methods	
Average Method Complexity	
Efferent couplings	
Afferent couplings	

30 percent of instances (222 instances) were left for testing of the model. Validation was performed by unseen data from Eclipse and AgroUML.

We looked at 5 distinct versions of Apache Ant for the within-project circumstances i.e., 1.3, 1.4, 1.5, 1.6, 1.7. All versions of the project are used for training of the model. There were 1692 instances in Apache Ant project wholly versions. There were 1184 samples records of training (70 percent) and 507 sample records of testing (30 percent) in the dataset.

The Apache Ivy version 2 dataset is used for validation purposes in cross-project prediction. This project is a novel project that has less historical information. In the entire project (in all versions) there were almost 16257 instances. For training, we used 70 percent data (11380 instances) and for testing of the model 30 percent data (4877 instances) are used.

The model's accuracy is also affected by the number of buggy samples in the database. The is-buggy is a Boolean attribute that specifies if, in a certain version, a file is a buggy (it is stated as ''1'') or not (stated as ''0''). In our datasets, there must be an equal number of true and false samples. It's quite difficult to create a training dataset with a balanced amount of buggy and non-buggy samples. The number of instances reported as buggy in all versions of all projects was 34.56 percent.

E. DIFFERENT TYPES OF METRICS USED FOR BUG PREDICTION

Several of the significant source code and code smell metrics that were evaluated in the study are summarized in subsequent Table 2.

F. PERFORMANCE EVALUATION METRICS

The performance is checked by passing through several parameters which are Confusion Matrix, Precision, Recall, F-Measure, ROC curve, PR curve, and Accuracy. This study considered two crucial factors: performance and effectiveness.

1) CONFUSION MATRIX

A typical machine learning approach for measuring the quality of an algorithm is to cross-classify predicted and real decision classes in a confusion matrix as well identified as an error matrix [67]. The ideal choice for calculating the accuracy and other measuring metrics of RF, NB, LR, SVM, and most of the classifiers is the Confusion matrix. A confusion matrix is a table that has the amount of correct and incorrect predictions produced by a classification model for the task of binary classification. It creates a table with all of a classifier's predicted and actual values. A classification model provides four different prediction outputs.

- True positive (TP): Malignant instances predicted as malignant via the ML model.
- False positives (FP): Benign instances predicted as malignant via the ML model.
- True negative (TN): Benign instances predicted as benign by via ML model.
- False negative (FN): Malignant instances were predicted as benign via my ML model.

Based on the above prediction results, various evaluation metrics have been presented in the literature.

2) ACCURACY

Accuracy is a parameter for evaluating the classification model. The number of correct predicted values multiplied by the whole number of predicted values is called accuracy. Equation (1) is also used to calculate accuracy in binary classification [68].

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{7}$$

where ''TP'' is a short form for True Positives, ''TN'' is short for True Negatives, ''FP'' is the short form for False Positives, and ''FN'' is the short form for False Negatives.

3) THE PRECISION

The function of related instances amongst the obtained instances is called precision. The following equation can be used to calculate it.

$$precision = \frac{TP}{TP + FP} \tag{8}$$

4) THE RECALL

The recall is the ratio accurately positive prediction for everybody in the actual consequence; the equation can be used to calculate it.

$$Recall = \frac{TP}{TP + FN} \tag{9}$$

5) THE F1-SCORE

A harmonic means of precision and recall would be the F1-Score. The F1-Score is the average of Precision and Recall when false positives and false negatives are considered. When the data distribution is imbalanced, the F1-Score is more effective than accuracy. F1-Score can be calculated by the Equation below.

$$F1 - score = 2 * \frac{\text{Pr}ecision * Recall}{\text{Pr}ecision + Recall} \tag{10}$$

6) ROC CURVE

The ROC curve shows how well a classification model works throughout the classification thresholds. In imbalance datasets, the AUC and F-measure are typically used to assess classifiers. The area under the ROC curve, which lies between [1, 0], measures the comparative performance of TPR and false positive rate (FPR).

This curve displays two values:

- True Positive Rate
- False Positive Rate

There is no difference between recall and TPR, which means recall is the same as TPR. TPR is defined as bellows:

$$TPR = \frac{TP}{TP + FN} \tag{11}$$

The following is how the False Positive Rate is defined:

$$FPR = \frac{PF}{FP + TN} \tag{12}$$

TPR & FPR at several classification thresholds are contrived on an ROC curve. As the classifying threshold is let down, more substances are classified as positive, leading to an upsurge in both False Positives and True Positives. AUC is an efficient, sorting-based technique that can compute the points in a ROC curve.

7) AREA UNDER ROC CURVE (AUC)

The AUC of ROC is one of the most important metrics used to measure classifier performance. ROC is a graphical tool that is used for binary classifiers' performance assessment. FPR and TPR can be combined into a single metric. TPR and FPR are calculated with separate thresholds and then plotted into a graph, with FPR values on the abscissa and TPR values on the ordinate. The produced curve is termed the ROC curve, and the metric we take into consideration is the Area Under the Curve (AUC). It should be remembered that the better the model, the greater the AUC.

8) PRECISION-RECALL CURVE (PR)

The PR curve displays the trade-offs between recall and precision for various thresholds. Having high accuracy showing a reduced FPR and excellent recall showing a low false-negative ratio, a big area under the curve suggests excellent recall and precision. On an imbalanced dataset, the PR curve is more informative than the ROC curve when assessing binary classification because of the usage of TN in FPR.

TABLE 3. Apache Ant 1.7 within version model.

Measure	k-NN	NB	LR	SVM	RF
Data division	90% and 10%				
Training Accuracy	92%	98.1%	100%	100%	100%
Testing Accuracy	89%	97.4%	99.1%	100%	100%
Balanced Accuracy	70%	97.2%	99.0%	100%	100%
Precision	1.00	0.95	0.98	1.00	1.00
Recall	0.38	1.00	1.00	1.00	1.00
F1 Score	0.55	0.97	0.99	1.00	1.00
Percentage of buggy instances in the dataset	22.28%				
Percentage of clean instances in the dataset	77.71%				

Precision vs. Recall is plotted on the PR curve. PR curve is a graphical tool that is used for binary classifiers performance comparison. Sometimes PR curve is further suitable than ROC. The ROC gives an idea of how the classifier overall acts, and it considers equally the positive and negative classes. PR curve is better for imbalanced data because it does not consider "True Negative", it measures the balance between two classes. The visual representation of the curves is a significant difference between ROC space and PR space. Viewing PR curves can reveal differences between algorithms that are not visible in the ROC space.

IV. EXPERIMENTAL RESULTS AND DISCUSSIONS

A. SYSTEM SPECIFICATION

The system that is used for simulations is an HP Intel core i5 4th generation desktop with 8GB RAM and 250 GB SSD and 500 GB HDD, 2.6 GHz processor, and Windows 10 64-bit operating system. Python 3.9 is used for smell-aware bug classification simulation. Jupyter Notebook is used for code execution. The library that was used is Scikit-Learn, Seaborn, and Matplotlib for Machine Learning.

B. DATASET DIVISION

The dataset is divided into several train and test ratios, as shown below:

- 90 percent training, 10 percent testing
- 70 percent training, 30 percent testing
- 60 percent training, 40 percent testing

C. RESULTS

The results of the proposed models are discussed briefly in this section. Furthermore, they demonstrate the evaluation outcomes of the bug classification model against various evaluation metrics. The experiment for smell-aware bug classification was performed using the PROMISE bug prediction dataset, and the different classification consequences obtained for various classifiers are shown in Table 4.

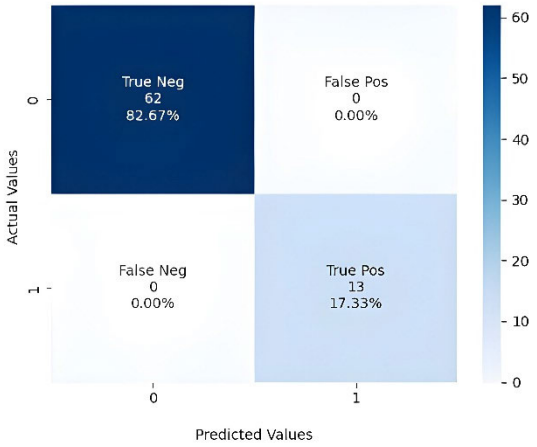


FIGURE 9. Confusion matrix for buggy vs non-buggy data on 90-10 % Ratio.

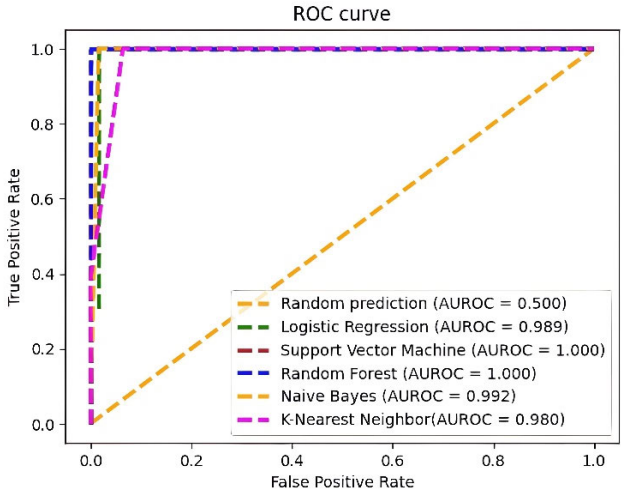


FIGURE 10. ROC curve within the version on 90-10 % ratio of the dataset.

D. PROPOSED MODEL CONFIGURATION AND RESULT

Using the PROMISE bug prediction dataset for training, testing, and evaluation of the model. The process of training, testing, and evaluation of the model on different datasets is shown in Figure 1. Following sub-sections discussed the entire experimentation.

E. EXPERIMENT WITHIN A VERSION

In a specific version of experimentation, the dataset is divided into several trains and test ratios, and then the model is trained accordingly.

1) FIRST EXPERIMENT WITHIN A VERSION FOR (90-(10) % RATIO

We experimented on Apache Ant version 1.7 for buggy and non-buggy classes, which is the PROMISE repository project dataset for bug classification. In the first step, the dataset is divided into two parts, 90% for the training of the model and

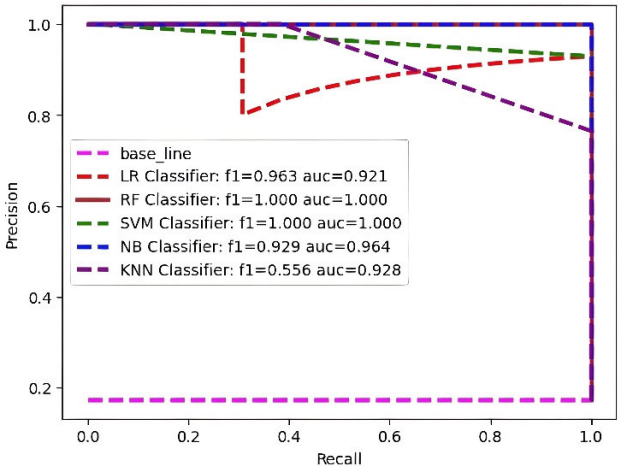


FIGURE 11. Precision-recall curve of 90% 10% Dataset division.

TABLE 4. Apache Ant 1.7 within version model for 70-30% dataset.

Measure	k-NN	NB	LR	SVM	RF
Data division	70% and 30%				
Training Accuracy	91%	98.2%	100%	100%	100%
Testing Accuracy	89%	97.1%	99.7%	100%	100%
Balanced Accuracy	77%	98%	99.7%	100%	100%
Precision	1.00	0.92	0.98	1.00	1.00
Recall	0.54	1.00	1.00	1.00	1.00
F1 Score	0.70	0.96	0.99	1.00	1.00
Percentage of buggy instances in the dataset				22.28%	
Percentage of clean instances in the dataset				77.71%	

10% for testing of the model. The comparison is done on five different ML classifiers, SVM, RF, LR, NB, and k-NN classifiers. In these classifiers, the RF and SVM classifiers have given the best result and the k-NN classifier gives the worst output. RF and SVM have the same output of evaluation metrics.

In all trained models, the confusion matrix of the best model is illustrated. Figure 9 demonstrates the confusion matrix of the buggy and non-buggy classes. In the columns, the predicted class is signified, whereas the actual class is signified in rows. The number of true and false predictions formed by the SVM classifier is shown in a confusion matrix in Figure 9.

It can be used to specify performance indicators like accuracy, precision, recall, and F1-score. TPR (True Positive Rate) and FPR (False Positive Rate) are used to evaluate the

efficiency of the proposed model. Numbers on the matrix diagonal designate the correct predictions, while values outside the matrix diagonal designate incorrect predictions. In brief, both the RF and SVM models have the same accuracy, correctly classifying the code as buggy and non-buggy classes with 100% accuracy.

2) PERFORMANCE ASSESSMENT OF SMELL-AWARE BUG CLASSIFICATION ON RF, SVM, LR, NB, AND K-NN CLASSIFIERS CONCERNING ROC CURVE FOR BUGGY AND NON-BUGGY CLASSES

The ROC and PR curves designate the performance of a classification model at a 90-10% ratio for bug classification. ROC and PR curves are two graphical tools that are used for comparison in binary classification. Figure 10 is a combined ROC curve for all the evaluated models in the ROC curve. AUC indicates how well the model can distinguish between classes. The PR curve shows the trade-off between precision. Figure 11 represents the PR curve, where the recall is on the x-axis and the precision is on the y-axis. Good recall is correlated with a low FN rate, while high precision is correlated with a low FP rate, this represents that SVM and RF models are very precise, and these algorithms did the best classification of codes as buggy or not buggy. Various performance metrics of the SVM, RF, NB, LR and k-NN classifiers for the dataset Apache Ant version 1.7 are shown in Table 3. As presented in Table 4, SVM and RF algorithms achieved a high accuracy rate of 100. The LR classifier gives 99% accuracy and the accuracy of the k-NN classifier is 89%. RF and SVM models showed an equal F1_score value, i.e.,1.0 whereas the LR model revealed the F1_score value of 0.99 and the k-NN classifier displayed the lowest value of 0.55 among all classifiers.

The ROC curve is constructed from two parameters:

1. True Positive Rate
2. False Positive Rate

As shown in Figure 10, every classifier has its own ROC curve. Each ROC space is specified by TPR (also called sensitivity) and FPR (also called specificity) as y and x-axis. The optimal prediction model would provide a point at coordinate (0,1) in the upper left corner of the ROC space, corresponding to 100% sensitivity (no false negatives) and 100% specificity (no false positives). In Figure 11, the RF and SVM classifiers have 100% sensitivity and 100% specificity, and they have huge AUC of ROC. So, RF and SVM algorithms did perfect classification, which means these models are the best than others. It should be noted that the k-NN model has the worst ROC curve.

In Figure 10 the PR curve is plotted and this is another good evaluation metric for imbalanced data. The AUC-PR curve is optimal for SVM and RF models.

3) SECOND EXPERIMENT WITHIN A VERSION

The bug prediction experimentation was conducted on the Apache Ant version 1.7 dataset for both buggy and non-buggy classes. In the first step, the dataset was divided into two parts: 70% for training the model and 30% for testing the model. The comparison was performed using five different ML classifiers: SVM, RF, LR, NB, and k-NN classifiers. Among these classifiers, the RF and SVM classifiers yielded the best results, while the k-NN classifier produced the worst output.

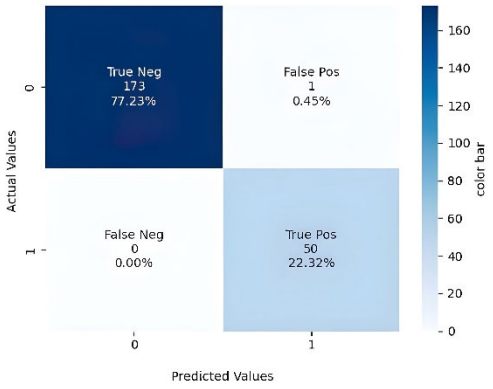


FIGURE 12. Confusion Matrix for Buggy vs non-buggy on 70-30% Ratio.

Figure 12 demonstrates the confusion matrix of the LR classifier for the buggy and non-buggy classes. In the column, the predicted class is signified while in the row the actual class is signified. It can be used to specify performance indicators like accuracy, precision, recall, F1-score, TPR, and FPR to evaluate the efficiency of the proposed model. Numbers on the matrix diagonal designate correct prediction, but values outside the matrix diagonal designate incorrect prediction. Figure 12 demonstrates the confusion matrix of the buggy and non-buggy classes. Out of 174 values 173 values of 0-class are predicted truly and 1 positive value is predicted wrongly. Correspondingly, 50 out of 50 values of 1-class are predicted accurately and did not predict any negative values incorrectly.

a: PERFORMANCE ASSESSMENT OF SMELL-AWARE BUG CLASSIFICATION ON RF, SVM, LR, NB, AND K-NN CLASSIFIERS ROC AND PRC FOR BUGGY AND NON-BUGGY

The performance of a classification model at a 70-30% ratio for bug classification level is described by the ROC and PR curves. ROC and PRC are two evaluation tools that are used for performance assessment of binary classifiers. Figure 13 is a combined ROC curve for all the evaluated models. It designates how well the model can differentiate between classes. As shown in Figure 13, every classifier has a particular ROC curve. Each ROC space is visualized by TPR and FPR as y and x-axis. In Figure 13, RF and SVM classifiers have 100% sensitivity and specificity, and they have huge AUC. So, RF and SVM algorithms did better classification, which means these models are better than others. On the other hand, the LR classifier AU-ROC value is outstanding, i.e.,0.997. The k-NN model AUC of the ROC curve is not glowing at all.

In Figure 14 the PR curve is plotted, in which the SVM and RF models AUC-PRC is 1, which denotes a high precision

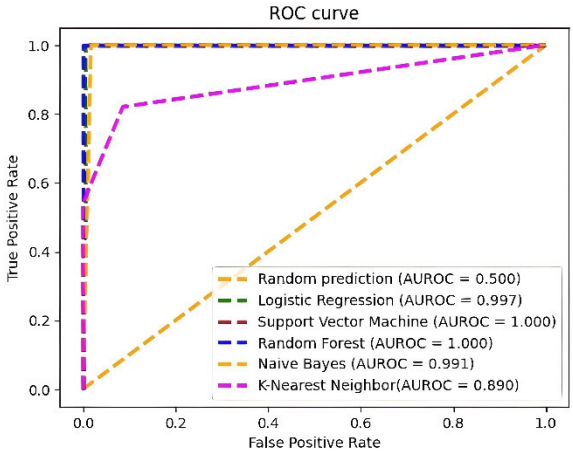


FIGURE 13. ROC curve within a version of the dataset on 70-3a 0 % ratio.

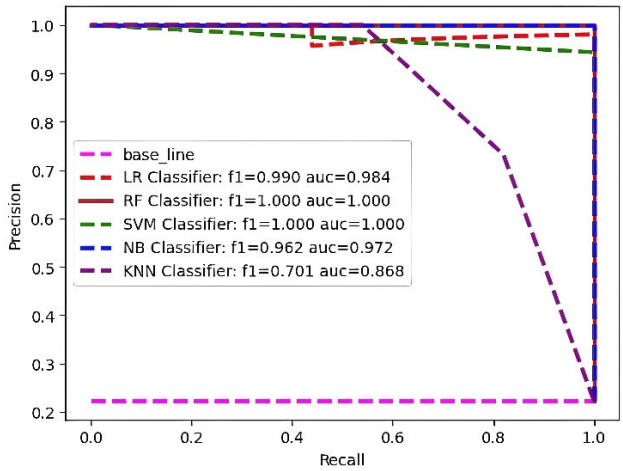


FIGURE 14. Precision-recall curve of 70% 30% Dataset division.

and a high recall, AUC-PRC is another good evaluation metric for imbalanced data. High recall corresponds to a low FN rate and high precision to a low FP rate, this represents that SVM and RF models are very precise, and these algorithms did the best classification of codes as buggy or non-buggy. The AUC-PR curve value is 0.98 for the LR classifier. The k-NN classifier's AUC-PR value is 0.86.

Table 4 signified other performance assessment parameters such as accuracy, precision, recall, and F1_score. Which is assessed for bug classification as buggy and non-buggy classes. As Table 4 demonstrated all trained models have given very good output except the k-NN classifier, the main reason is the integration of the code smell metrics with source code metrics in the given dataset. As shown in Table 5, the SVM and RF algorithms give very precise outcomes with both Precision and Recall being 1.00. It is also memorable that the LR classifier has given a precision value of 0.98, and the recall value is 1.00. But the k-NN classifier recall value is low i.e., 0.54 and the F1_score is 0.7 among all algorithms.

TABLE 5. Within version performance evaluation metrics.

Measure	k-NN	NB	LR	SVM	RF
Data division	50% and 50%				
Training Accuracy	90 %	98.1 %	100%	100%	100%
Testing Accuracy	89 %	98.6 %	99.65%	100%	100%
Balanced Accuracy	73 %	98.6 %	99.6%	100%	100%
Precision	1.00	0.98	0.99	1.00	1.00
Recall	0.46	0.98	0.99	1.00	1.00
F1 Score	0.63	0.98	0.99	1.00	1.00
Percentage of buggy instances in the dataset				22.28%	
Percentage of clean instances in the dataset				77.71%	

4) THIRD EXPERIMENT WITHIN A VERSION

This experimentation is done for buggy vs non-buggy classes. This experiment is done for a 50-50% ratio.

Figure 16 demonstrates the confusion matrix of the NB model for the buggy and non-buggy classes. The predicted class is indicated in the column whereas the actual class is shown in the row. Numbers on the matrix diagonal designate correct prediction, but values outside the matrix diagonal designate incorrect prediction. Figure 15 demonstrates the confusion matrix of the buggy and non-buggy classes, 294 out of 300 values of 0-class (non-buggy) are predicted truly and 6 positive value is predicted wrongly. Correspondingly, 73 out of 73 values of 1-class are predicted accurately, while no value is predicted incorrectly.

a: Performance assessment of Smell-aware bug classification on RF, SVM, LR, NB, and k-NN classifiers ROC and PRC for buggy and non-buggy

The performance of a classification model at a 50-50% ratio for buggy and non-buggy classes is defined by the ROC curve and PR curve. Figure 16 is a combined ROC curve for all the evaluated models, in which each ROC space is stated by TPR and FPR as y and x-axis. SVM and RF algorithms' recall and cut-off are always better than LR, NB, and k-NN algorithms. RF and SVM classifiers have 100% sensitivity and specificity, and these algorithms have vast AUC. On the other hand, the LR classifier AU-ROC value is outstanding, i.e.,0.98. The AUC-ROC for the NB model is 0.97 and k-NN AUC-ROC is 0.86. The PR curve is plotted in Figure 17, in which the SVM and RF model's AUC-PR curve value is 1,

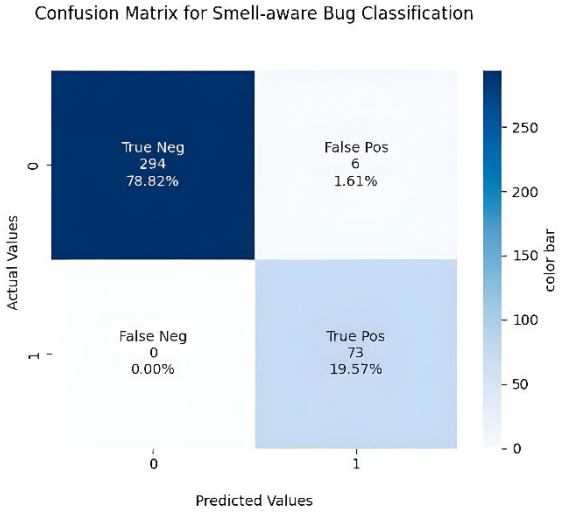


FIGURE 15. Confusion matrix for Buggy vs non-buggy on 50-50% ratio.

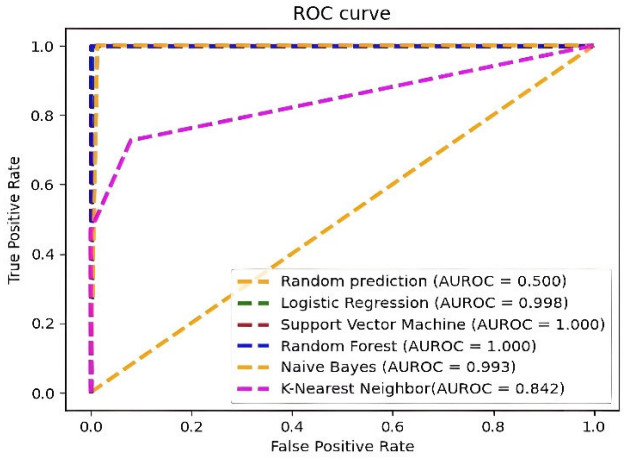


FIGURE 16. ROC curve within a version of the dataset on a 50-50 % ratio.

which denotes high precision and high recall. This signifies that SVM and RF models are very precise. The LR classifier value of the AUC-PR curve is 0.98 and the k-NN classifier's AUC-PR value is 0.86.

Table 5 shows the complete performance report of the k-NN, NB, LR, SVM, and RF models. In this experiment SVM and RF give equal performance assessment metrics values. The overall best accuracy of the model is 100 % on SVM and RF classifiers for a 50-50% ratio, as shown above in table 6. RF and SVM models demonstrate equal F1_score value, i.e.,1.0 whereas the LR model revealed the F1_score value of 0.99 and the k-NN classifier having a low F1_score value of 0.63. The k-NN classifier gives a better F1_score value in the (50-50) % division of the dataset than the (70-30) % dataset division as train and test samples.

F. WITHIN THE PROJECT

This part concealments the bug prediction in the context of the within-project and specifically does not give evidence on how the model accomplishes within a single version of the

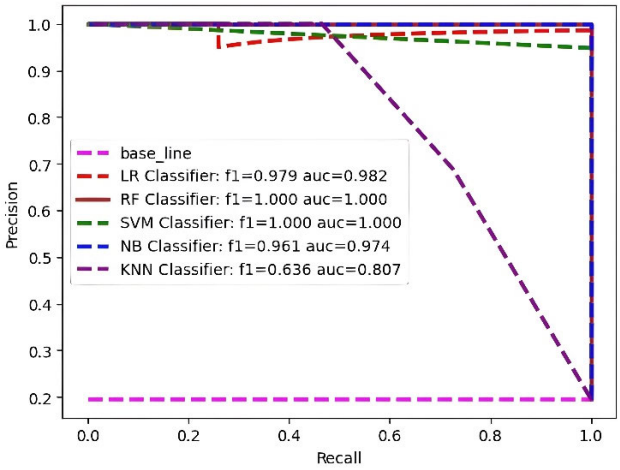


FIGURE 17. Precision-recall curve of 50% 50% dataset division.

TABLE 6. Apache Ant within-project model measures.

Measure	k-NN	NB	LR	RF	SVM
Data division	70% and 30%				
Training Accuracy	99.7%	98%	100%	100%	100%
Testing Accuracy	94.4%	96%	100%	100%	100%
Balanced Accuracy	94.7%	96%	100%	100%	100%
Precision	0.923	0.93	1.00	1.00	1.00
Recall	0.972	1.00	1.00	1.00	1.00
F1 Score	0.947	0.96	1.00	1.00	1.00
Percentage of buggy instances in the dataset					19.42%
Percentage of clean instances in the dataset					80.57%

project. The outcome of the Apache Ant project is given in Table 6.

As findings demonstrated in Table 7 conclude, generally all models have shown a good performance of the evaluation metrics, and RF, SVM, and LR algorithms provide the most accurate result within the project and give an equal output of the performance evaluation metrics. The main reason can be the dataset is balanced by SMOTE Technique and also the number of samples is more than within a version, and the dataset has a good number of buggy instances.

The F2-measure, which is the chosen metric to assess the model is excessively high for all the classifiers, we have seen improvements in the performance of some models (see Table.7) both sensitivity and F2-measures values have improved suggesting a better distinction between the two classes and decreases of the false positive's predictions. The

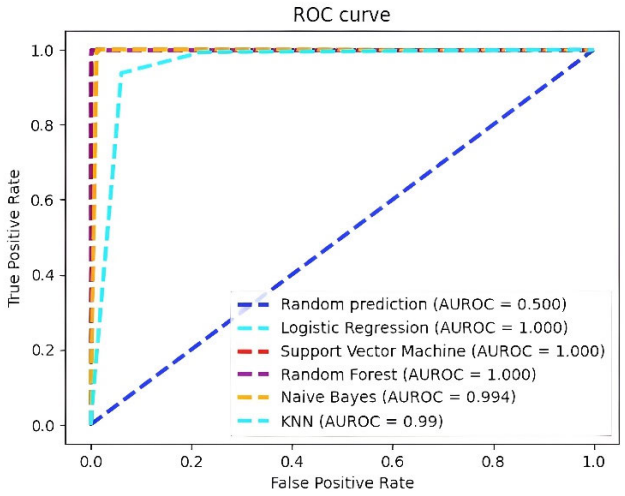


FIGURE 18. ROC curve for within-project.

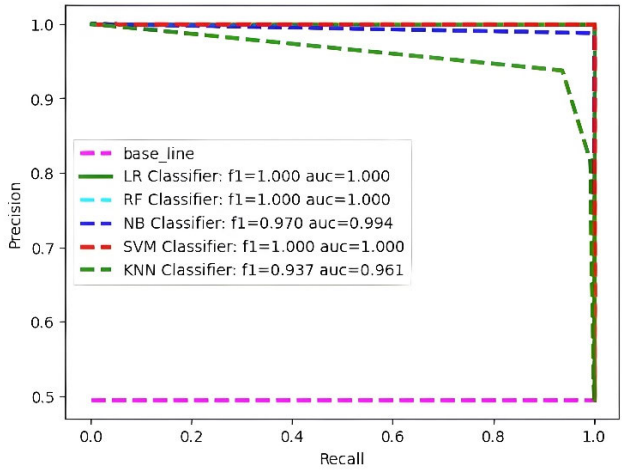


FIGURE 19. Precision-recall curve for within-project.

performance of a classification model within the project for buggy vs non-buggy classes is depicted by the ROC curve in Figure 18. The PR curve is drawn in Figure 19, in which the SVM, RF, and LR model's AUC-PR curve value is 1, which denotes high precision and high recall. This signifies that SVM, RF, and LR models are very precise.

The AUC-PR curve value is 0.99 for the NB classifier. The k-NN classifier's AUC-PR value is 0.96, which illustrates that k-NN gives better results within the project than within a version. shown in the results, e.g., k-NN classifier in 14 datasets has the range of 73% to 97%. This can occur due to sample overlapping, noise interference, and blindness of neighbor selection during balancing and the size of the dataset also have a huge impact on the training of the model. For instance, the Apache Forrest dataset has a total of 61 samples which is insufficient to train the model accurately. Notable that the performance evaluation parameters for some data sets are extremely high, for example, the Apache Synapse and Apache Camel dataset's performance evaluation matrix is very high, the main reason can be enough samples in the dataset for good training of the model.

TABLE 7. Apache Ant across project model metrics evaluation report.

Measure	k-NN	LR	NB	RF	SVM
Data division	70% and 30%				
Training Accuracy	92%	99.67%	80 %	100%	99.7%
Testing Accuracy	85%	99.67%	81 %	99.7%	99.7%
Balanced Accuracy	85%	99.67%	81 %	99.73%	99.73%
Precision	0.84	0.99	0.9 0	0.99	0.98
Recall	0.87	0.99	0.7 0	1.00	1.00
F1 Score	0.85	0.99	0.7 9	0.99	0.994
Percentage of buggy instances in the dataset					34.56%
Percentage of clean instances in the dataset					65.43%

For instance, the Apache Forrest dataset has a total of 61 samples which is insufficient to train the model accurately. Notably, the performance evaluation parameters for some data sets are extremely high, for example, the Apache Synapse and Apache Camel dataset's performance evaluation matrix is very high, the main reason can be enough samples in the dataset for good training of the model.

V. ACROSS THE PROJECTS

From all 14 projects, all the versions were used for training the model to achieve the results. Table 9 is the evaluation result of the designed models.

For the validation purpose of the trained model, the second version of Apache Ivy is used. In the training dataset, there were 16257 samples, and the buggy samples were 34.56% (5620 instances). However, in the cross-project prediction model, there is 34% of buggy samples in the training set, and therefore, the accuracy of cross-project prediction is low as compared to within the version in the project prediction.

In this experiment, the NB algorithm has given poor results as compared to the k-NN algorithm. Moreover, within the version and the project NB algorithm has given better results than k-NN. For a better understanding and illustration of the models, the following are the ROC and PR curves.

Figure 20 is a combined ROC curve for all the trained models across the projects. Each ROC space is stated by TPR and FPR as y and x-axis. In figure 20, the RF classifier ROC space is 1 and the ROC curve is greater than other classifiers.

TABLE 8. The comprehensive result of the project.

Datasets	Algorithms	Findings			
		Acc	Precision	Recall	F1 Score
Apache Ant	k-NN	91.4%	0.86	0.97	0.91
	NB	96%	0.93	1.00	0.96
	LR, SVM, RF	100%	1.00	1.00	1.00
Apache Camel	k-NN	90%	0.85	0.96	0.90
	NB	98.6%	0.97	1.00	0.98
	LR, SVM, RF	100%	1.00	1.00	1.00
Apache Forrest	k-NN	75.7%	0.69	1.00	0.81
	NB	93%	0.9	1.00	0.94
	LR, SVM, RF	100%	1.00	1.00	1.00
Apache Ivy	k-NN	93.6%	0.87	1.00	0.93
	NB	99%	0.98	1.00	0.99
	LR, SVM, RF	100%	1.00	1.00	1.00
Apache Log4j	k-NN	77%	0.82	0.54	0.68
	NB	95%	0.90	1.0	0.94
	LR	98%	0.95	1.0	0.97
	SVM, RF	100%	1.0	1.0	1.0
Apache Lucene	k-NN	75.8%	0.90	0.55	0.68
	NB	98.8%	0.96	1.0	0.98
	LR, SVM, RF	100%	1.0	1.0	1.0
Apache Pbeans	k-NN	75.4%	0.6	0.9	0.72
	LR, NB	96%	0.90	1.0	0.95
	SVM, RF	100%	1.0	1.0	1.0
Apache POI	k-NN	87%	0.9	0.83	0.86
	NB	98.3%	0.96	1.0	0.98
	LR	100%	0.90	1.0	0.95
	SVM, RF	100%	1.0	1.0	1.0
Apache Synapse	k-NN	91.1%	0.87	0.95	0.91
	NB	99%	0.98	1.0	0.99
	LR, SVM, RF	100%	1.0	1.0	1.0
Apache Tomcat	k-NN	97.5 %	0.94	1.0	0.96
	NB	99.2%	1.0	0.98	0.99
	LR	97.7%	0.96	0.98	0.97
	SVM	98.3%	0.96	1.0	0.98
	RF	99.6%	0.99	1.0	0.99
Apache Velocity	k-NN	73.7%	0.68	0.84	0.75
	NB	72%	0.72	0.55	0.62
	LR	76.9%	0.75	0.76	0.75
	SVM	86%	0.82	0.92	0.87
	RF	99%	0.98	1.0	0.99
Apache Xalan	k-NN	86%	0.96	0.75	0.84
	NB	97%	0.96	1.0	0.98
	LR, SVM, RF	100%	1.0	1.0	1.0
Apache Xerces	k-NN	78%	0.82	0.74	0.78
	NB	98%	0.97	1.0	0.98
	LR	99.6%	1.0	0.99	0.99
	SVM, RF	100%	1.0	1.0	1.0
Apache jEdit	k-NN	89%	0.83	0.97	0.90
	NB	97%	0.95	1.0	0.97
	LR, SVM, RF	100%	1.0	1.0	1.0

TABLE 9. Comparison of our proposed model with other classifiers in the literature.

Method	Dataset	Observed Accuracy	Precision	Recall	F1 Score	AUC-ROC	AUC-PR
SVM [15]	Android, Mozila	80%	N/A	N/A	0.9	N/A	N/A
RF, SLC [10]	Kaggle Bug Dataset	77%, 87%	0.71, 0.81	0.33, 0.87	0.450.83	N/A	N/A
CNN-H-A-P [69]	PSC	98%	0.991	0.99	N/A	N/A	N/A
LR [22]	Apache Xalan	100%	1.00	1.00	1.00	1.00	N/A
	Apache Log4j	95 %	1.00	0.98	0.99	0.93	N/A
	Apache Tomcat	79%	0.20	0.31	0.24	0.94	N/A
RF [1]	APACHE ANT	87%	0.88	0.40	0.55	N/A	N/A
	Apache Xalan	100%	1.00	1.00	1.00	N/A	N/A
Proposed model: LR, SVM, RF, SVM, RF	Apache Tomcat	98.3%	0.96	1.0	0.98	N/A	N/A
	Apache Xalan	100%	1.00	1.00	1.00	1.00	1.00
	Apache Ant	100%	1.00	1.00	1.00	1.00	1.00
	Apache Log4j	100%	1.00	1.00	1.00	1.00	1.00

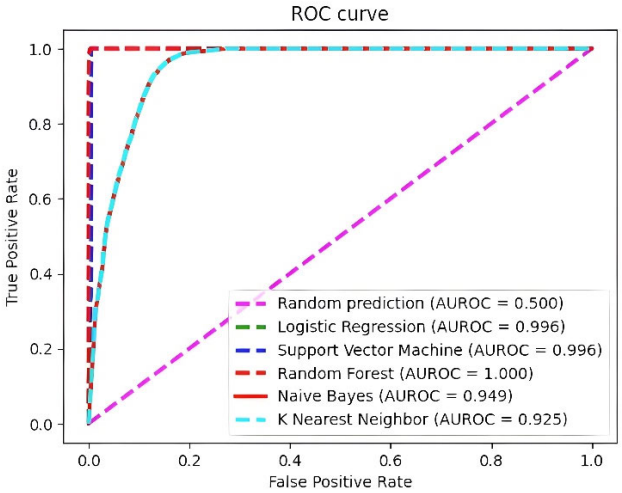


FIGURE 20. ROC curve across the project.

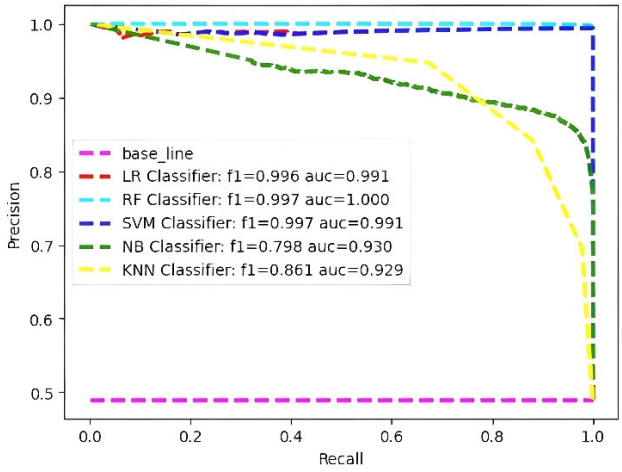


FIGURE 21. Precision-recall curve across the project.

Therefore, RF algorithm recall and cut-off are always better than LR, SVM, NB, and k-NN algorithms across the project prediction. RF classifier has 100% sensitivity and % specificity and it has massive AUC. On the other hand, the LR classifier AU-ROC value is outstanding, i.e.,0.988.

In Figure 21, the PR curve is contrived, showing that the RF model AUC-PR value is 1, which denotes high precision and high recall. So, this implies that the RF model is very accurate. The AUC-PR curve value is 0.988 for the LR classifier. The k-NN classifier's AUC-PR value is 0.845.

A. COMPARISON OF THE PROPOSED MODEL WITH EXISTING MODELS:

This part presents a performance comparison of the proposed model with the existing models; therefore, we selected five ML classifiers. These classifiers are NB, RF, SVM, k-NN, and LR.

This study explores various evaluation metrics of the five models within the version, within the project, and across the projects. The result of this comparison in terms of accuracy, precision, recall, F1 score, AUC-ROC, and AUC-PR are listed in table 9.

The result of this assessment is presented in table 9 that the proposed classifiers exceed the existing classifiers in terms of all six-measurement metrics.

VI. LIMITATIONS OF THE STUDY

In the field of software engineering various tasks can be formulated as learning problems and can be solved using machine learning algorithms. However, in the Software engineering domain the source of training data is source code, and the majority of the datasets are based on JAVA code. Therefore, in this study, only software systems coded in the Java programming language are evaluated for code smell

prediction. Moreover, the scope of our smell extraction was restricted to open-source Java projects exclusively from the Apache repository, this limited selection could potentially impact the generalizability of our results.

VII. CONCLUSION AND FUTURE WORK

Software defects are called bugs in a software development process - unanticipated deeds and actions figured out by quality control engineers during application testing and are preserved as software bugs. Bugs have high effects on software quality. The process of bug fixing is exceptionally steady and time-consuming. Therefore, it is crucial to detect bugs automatically.

The primary goal of this investigation is to create a smell-aware bug prediction model by using code smell as a nominee metric. To be smell-aware, we added an intensity index to the dataset. The results showed that using the intensity index as a predictor for bug prediction improves the accuracy of the bug prediction model. Furthermore, the data show that the severity index is more significant than any other quality metric in predicting the bug-proneness of the smelly classes. The findings suggest that using the severity index as a reliable indicator of buggy modules improves the effectiveness of structurally based baseline models for bug prediction. Furthermore, they also emphasize the significance of the intensity of code smells in the process metrics-based prediction approaches."

We provided empirical evidence in this study that code smell-based metrics are quite useful in bug prediction. Using several source code metrics and code smell-based metrics proposed in the literature, we constructed a bug prediction model. To create the model, we employed k-NN, NB, RF, SVM, and LR algorithms. Multiple versions of fourteen different open-source projects were used to train the bug prediction model. We experimented with how our bug prediction model behaved within the version, within the project, and across the projects.

To emphasize the following are the main conclusion from our research:

- Using only source code metrics to anticipate project issues is insufficient.
- When code smell-based metrics are combined with source code metrics, accuracy, and F1 score can be improved.
- When compared to other algorithms, RF and SVM algorithms have demonstrated the best results in terms of accuracy.
- The presence of a large amount of numerical/categorical data, as well as training with a growing number of samples, might be the primary factors behind Random Forest's superior performance.
- The main reason behind the good results of the SVM algorithm might be that it is used effectively for slightly large and complex linear and non-linear datasets.
- Our features are not independent of each other, which is why Naive Bayes did not perform well in the study.
- Code smell-based metrics can be used to accurately forecast bugs across projects. When there are fewer buggy cases in the system, we were able to get more accurate findings.

In future work, we would like to evaluate the performance assessment of the model in other programming languages as well. Furthermore, we will undertake additional research on the attributes of the Intensity index in the context of multi-class bug classification based on the samples collected from different languages from within a project, within a version, and across the project's data set.

APPENDIX A

Code is available on GitHub https://github.com/hqsikandar/Bug-prediction-train-model-by-Dr-Sikandar-Ali-and-Khyber.

ACKNOWLEDGMENT

The authors acknowledge the support of their respective universities. *(Khyber and Sikandar Ali are co-first authors.)*

REFERENCES

[1] G. M. Ubayawardana and D. D. Karunaratna, "Bug prediction model using code smells," in *Proc. 18th Int. Conf. Adv. ICT for Emerg. Regions (ICTer)*, Sep. 2018, pp. 70–77.
[2] N. K. Nagwani and P. Singh, "Bug mining model based on event-component similarity to discover similar and duplicate GUI bugs," in *Proc. IEEE Int. Advance Comput. Conf.*, Mar. 2009, pp. 1388–1392.
[3] N. K. Nagwani and S. Verma, "Predictive data mining model for software bug estimation using average weighted similarity," in *Proc. IEEE 2nd Int. Advance Comput. Conf. (IACC)*, Feb. 2010, pp. 373–378.
[4] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging: NIER track," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, May 2011, pp. 884–887.
[5] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model," *IEEE Trans. Softw. Eng.*, vol. 43, no. 3, pp. 272–297, Mar. 2017.
[6] R. Varshneya, "There's no such thing as a bug-free app," *Entrepreneur*, vol. 22, Oct. 2015.
[7] A. Hammouri, M. Hammad, M. Alnabhan, and F. Alsarayrah, "Software bug prediction using machine learning approach," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 11, no. 12, pp. 1401–1404, Dec. 2023.
[8] A. S. Cairo, G. D. F. Carneiro, and M. P. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, vol. 9, no. 11, p. 273, 2018.
[9] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Proc. 10th Int. Conf. Quality Softw.*, Jul. 2010, pp. 23–31.
[10] R. R. Panda and N. K. Nagwani, "Software bug categorization technique based on fuzzy similarity," in *Proc. IEEE 9th Int. Conf. Adv. Comput. (IACC)*, Dec. 2019, pp. 1–6.
[11] S. Gujral, G. Sharma, S. Sharma, and Diksha, "Classifying bug severity using dictionary based approach," in *Proc. Int. Conf. Futuristic Trends Comput. Anal. Knowl. Manage. (ABLAZE)*, Feb. 2015, pp. 599–602.
[12] A. Jayagopal, R. Kaushik, A. Krishnan, R. Nalla, and S. Ruttala, *WITHDRAWN: Bug Classification Using Machine and Deep Learning Algorithms*. Amsterdam, The Netherlands: Elsevier, 2021.
[13] S. J. Dommati, R. Agrawal, G. R. M. Reddy, and S. S. Kamath, "Bug classification: Feature extraction and comparison of event model using Naïve Bayes approach," 2013, *arXiv:1304.1677*.
[14] K. Pan, S. Kim, and E. Whitehead, Jr., "Bug classification using program slicing metrics," in *Proc. 6th IEEE Int. Workshop Source Code Anal. Manipulation*, Sep. 2006, pp. 31–42.

[15] N. K. Nagwani and S. Verma, "CLUBAS: An algorithm and Java based tool for software bug classification using bug attributes similarities," Tech. Rep., 2012.

[16] S. D. Immaculate, M. F. Begam, and M. Floramary, "Software bug prediction using supervised machine learning algorithms," in *Proc. Int. Conf. Data Sci. Commun. (IconDSC)*, Mar. 2019, pp. 1–7.

[17] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data," *Empirical Softw. Eng.*, vol. 19, no. 4, pp. 1111–1143, Aug. 2014.

[18] M. A. Mabayoje, A. O. Balogun, H. A. Jibril, J. O. Atoyebi, H. A. Mojeed, and V. E. Adeyemo, "Parameter tuning in KNN for software defect prediction: An empirical analysis," Tech. Rep., 2019.

[19] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 270–279.

[20] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 194–218, Feb. 2019.

[21] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *Proc. IEEE 7th Int. Workshop Manag. Tech. Debt (MTD)*, Oct. 2015, pp. 16–24.

[22] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2016, pp. 244–255.

[23] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proc. 9th Work. Conf. Reverse Eng.*, Oct. 2002, pp. 97–106.

[24] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, Apr. 1993.

[25] D. L. Parnas, "Software aging," in *Proc. 16th Int. Conf. Softw. Eng.*, May 1994, pp. 279–287.

[26] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code Addison-Wesley Professional*. Berkeley, CA, USA, 1999.

[27] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[28] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Eng.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.

[29] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018.

[30] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proc. 15th Eur. Conf. Softw. Maintenance Reengineering*, Mar. 2011, pp. 181–190.

[31] P. Team, "An exploratory study of the impact of code smells on software change-proneness," Tech. Rep.

[32] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 682–691.

[33] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. D. Nascimento, M. F. Freitas, and M. G. D. Mendonça, "A systematic review on the code smell effect," *J. Syst. Softw.*, vol. 144, pp. 450–477, Oct. 2018.

[34] F. Akiyama, "An example of software system debugging," *IFIP Congr.*, vol. 71, pp. 353–359, Jan. 1971.

[35] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. Amsterdam, The Netherlands: Elsevier, 1977.

[36] L. Hatton, "Conservation of information: Software'sHidden clockwork?" *IEEE Trans. Softw. Eng.*, vol. 40, no. 5, pp. 450–460, May 2014.

[37] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 5, pp. 423–433, May 1992.

[38] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[39] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 1988, pp. 35–46.

[40] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proc. 16th Work. Conf. Reverse Eng.*, Oct. 2009, pp. 75–84.

[41] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2009, pp. 390–400.

[42] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 1–39, Sep. 2014.

[43] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, Sep. 2010, pp. 1–10.

[44] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, "The promise repository of empirical software engineering data," Dept. Comput. Sci., West Virginia Univ., Morgantown, WV, USA, Tech. Rep., 2012.

[45] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 356–370, May 2011.

[46] T. Mende, "Replication of defect prediction studies: Problems, pitfalls and recommendations," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, Sep. 2010, pp. 1–10.

[47] A. Agresti, *An Introduction to Categorical Data Analysis*. Hoboken, NJ, USA: Wiley, 2018.

[48] D. G. Kleinbaum, L. Kupper, K. Müller, and A. Nizam, "Regression diagnostics," in *Applied Regression Analysis and Other Multivariable Methods*, vol. 2, 1998.

[49] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Cham, Switzerland: Springer, 2009.

[50] P. Karsmakers, K. Pelckmans, and J. A. K. Suykens, "Multi-class kernel logistic regression: A fixed-size implementation," in *Proc. Int. Joint Conf. Neural Netw.*, Aug. 2007, pp. 1756–1761.

[51] D. W. Hosmer and S. Lemeshow, *Applied Logistic Regression*. New York, NY, USA: Wiley, 2000.

[52] Y. Liu, Y. Wang, and J. Zhang, "New machine learning algorithm: Random forest," in *Proc. Int. Conf. Inf. Comput. Appl.* Chengde, China: Springer, Sep. 2012, pp. 246–252.

[53] K. Kirasich, T. Smith, and B. Sadler, "Random forest vs logistic regression: Binary classification for heterogeneous datasets," *SMU Data Sci. Rev.*, vol. 1, no. 3, p. 9, 2018.

[54] V. Vapnik, *The Nature of Statistical Learning Theory*. Cham, Switzerland: Springer, 1999.

[55] C. J. C. Burges, "A tutorial on support vector machines for pattern recognition," *Data Mining Knowl. Discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[56] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, U.K.: Cambridge Univ. Press, 2000.

[57] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.

[58] B. Schölkopf and A. J. Smola, "Support vector machines," in *Learning With Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*, 2001, pp. 187–188.

[59] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proc. 5th Annu. ACM Workshop Comput. Learn. Theory*, 1992, pp. 144–152.

[60] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 1–27, Apr. 2011.

[61] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.

[62] W. S. Noble, "What is a support vector machine?" *Nature Biotechnol.*, vol. 24, no. 12, pp. 1565–1567, Dec. 2006.

[63] A. Ng and M. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 14, 2001.

[64] P. Cunningham and S. J. Delany, "K-nearest neighbour classifiers—A tutorial," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 1–25, Jul. 2022.

[65] O. J. T. D. S. Harrison, *Machine Learning Basics With the K-Nearest Neighbors Algorithm*, vol. 11, 2018.

[66] D. Dagao and G. Yang, "Preprocessing technology of consuming big data based on user interest with Internet of location mining," *Int. J. Comput. Appl.*, vol. 43, no. 2, pp. 147–152, Feb. 2021.

[67] I. Düntsch and G. Gediga, "Confusion matrices and rough set data analysis," *J. Phys., Conf. Ser.*, vol. 1229, no. 1, May 2019, Art. no. 012055.

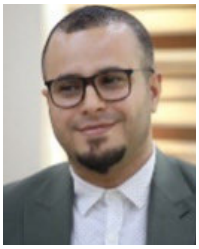
[68] Z. Imran, "Predicting bug severity in open-source software systems using scalable machine learning techniques," Tech. Rep., 2016.
[69] P. K. Chaubey and T. K. J. M. T. P. Arora, "Software bug prediction and classification by global pooling of different activation of convolution layers," Tech. Rep., 2020.

KHYBER received the B.Sc. degree in computer science from the Faculty of Computer Science, Nangarhar University, Afghanistan, in 2019, and the M.Sc. degree in computer science from The University of Haripur, Pakistan, in 2022. His research interests include software engineering, agile software development, software bug classification, machine learning, and deep learning.

SIKANDAR ALI received the Ph.D. and Postdoctoral degrees from the China University of Petroleum, Beijing, in 2019 and 2021, respectively. He is currently an Assistant Professor. He has authored more than 70 articles in highly cited journals and conferences. His research interests include business process management, anomaly detection in multi-sensor systems, software outsourcing partnerships, software testing and test automation, bug prediction, bug fixing, software incidence classification, source code transformation, agile software development, and global software engineering. He served as a technical program committee member for more than 20 conferences and acted as a reviewer for many well-reputed journals. He also organizes many special issues. He is currently an Academic Editor of *Mobile Information Systems*, *Journal of Computer Information Systems*, and *Scientific Programming*.

FAZLI WAHID received the B.S. degree in computer science from the University of Malakand, Pakistan, in 2006, the M.S. degree in computer science from SZABIST, Islamabad, Pakistan, in 2015, and the Ph.D. degree in computer science from University Tun Hussein Onn Malaysia, in 2020. He is currently an Assistant Professor with the Department of Information Technology, The University of Haripur, Pakistan. Previously, he was an Assistant Professor of computer science with The University of Lahore, Pakistan. His research interests include machine learning, deep learning, medical imaging, artificial intelligence, and related fields. Related to all these areas, he has published more than 40 research articles in well-reputed journals and conferences. His areas of interest are energy consumption prediction, optimization, and management using multilayer perceptron, artificial bee colony, ant colony, swarm intelligence, and other machine learning techniques.

SAMAD BASEER received the B.Sc. and M.S. degrees in computer system engineering from the University of Engineering and Technology, Peshawar, Pakistan, in 2004 and 2007, respectively, and the Ph.D. degree in information communication technology from the Asian Institute of Technology, in 2012. He is currently an Assistant Professor with the Department of Computer System Engineering, University of Engineering and Technology. His research interests include agile methodology in software engineering, current network security issues in next-generation networks, 5G quality of service parameters and constraints, and cooperative communications techniques in disaster-hit areas. He has acted as a referee and reviewed various journal articles along with conference papers. He has contributed to organizing several workshops and special sessions on emerging technologies which include wireless communication network security and software engineering.



AHMED ALKHAYYAT received the B.Sc. degree in electrical engineering from Al Kufa University, Najaf, Iraq, in 2007, the M.Sc. degree from the Dehradun Institute of Technology, Dehradun, India, in 2010, and the Ph.D. degree from Cankaya University, Ankara, Turkey, in 2015. He is currently the Dean of International Relationships and the Manager of the world ranking with The Islamic University, Najaf. His research interests include the IoT in healthcare systems, SDN, network coding, cognitive radio, efficient-energy routing algorithms, efficient-energy MAC protocol in cooperative wireless networks, wireless body area networks, and cross-layer designing for self-organized networks. He contributed to organizing several IEEE conferences, workshops, and special sessions. To serve his community, he acted as a reviewer for several journals and conferences.

AKRAM M. AL-RADAEI was born in Thamar, Yemen. He received the B.S. degree in information technology from Thamar University, Thamar, and the master's degree in software engineering from International Islamic University Islamabad (IIUI), Pakistan. He is currently pursuing the Ph.D. degree with Thamar University. He was a Faculty Member with Thamar University. His current research interests include software engineering, software testing, artificial intelligence applications and algorithms, machine learning, and deep learning.