Automated Software Engineering (2024) 31:70
https://doi.org/10.1007/s10515-024-00467-4

A holistic approach to software fault prediction with dynamic classification

S. Kaliraj[1] · Velisetti Geetha Pavan Sahasranth[1] · V. Sivakumar[1]

Received: 9 February 2024 / Accepted: 19 August 2024 / Published online: 4 September 2024

Abstract

Software Fault Prediction is a critical domain in machine learning aimed at pre-emptively identifying and mitigating software faults. This study addresses challenges related to imbalanced datasets and feature selection, significantly enhancing the effectiveness of fault prediction models. We mitigate class imbalance in the Unified Dataset using the Random-Over Sampling technique, resulting in superior accuracy for minority-class predictions. Additionally, we employ the innovative Ant-Colony Optimization algorithm (ACO) for feature selection, extracting pertinent features to amplify model performance. Recognizing the limitations of individual machine learning models, we introduce the Dynamic Classifier, a ground-breaking ensemble that combines predictions from multiple algorithms, elevating fault prediction precision. Model parameters are fine-tuned using the Grid-Search Method, achieving an accuracy of 94.129% and superior overall performance compared to random forest, decision tree and other standard machine learning algorithms. The core contribution of this study lies in the comparative analysis, pitting our Dynamic Classifier against Standard Algorithms using diverse performance metrics. The results unequivocally establish the Dynamic Classifier as a frontrunner, highlighting its prowess in fault prediction. In conclusion, this research introduces a comprehensive and innovative approach to software fault prediction. It pioneers the resolution of class imbalance, employs cutting-edge feature selection, and introduces dynamic ensemble classifiers. The proposed methodology, showcasing a significant advancement in performance over existing methods, illuminates the path toward developing more accurate and efficient fault prediction models.

Keywords Software fault prediction · Machine learning · Class-imbalance · Feature selection · Ant colony optimization · Hyperparameter tuning · Dynamic classifier · Performance evaluation

Extended author information available on the last page of the article

1 Introduction

In the world of software engineering, a "fault" refers to any flaw in a software's code also known as bugs or defects, which can manifest in various forms, including logical errors, coding mistakes, or incorrect functionality within the software. These faults can lead to system failures, crashes, or unexpected behavior, posing significant risks to businesses, users, and the overall software ecosystem. Software systems are complex, made up of thousands, or even millions, of lines of code, so it's not uncommon for faults to exist within them. These faults can sneak in during the development process due to human error, misunderstanding requirements, or not testing thoroughly enough. They can show up in various ways, like the software not working as expected, crashing, freezing, or even opening up security holes. Recognizing these faults and understanding how they occur is essential for making software systems reliable and secure. That's why software fault prediction has become such an important area of focus within the field of software engineering.

Software fault prediction (SFP) is a crucial area of research in software engineering (Failed 2020a). As software continuously evolves, studying software faults has become increasingly important in software reliability research (Cetiner and Sahingoz 2020). Software Fault Prediction is pivotal in ensuring quality control in software development. However, relying on outdated methods for fault prediction necessitates a substantial allocation of resources to uphold product quality throughout the software development life cycle (Arshad, et al. 2018; Chen et al. 2016; AlShaikh and Elmedany 2022). Early detection of faulty classes in the software development lifecycle can lead to substantial savings in terms of resources, time, and user satisfaction (Failed 2020b). The main goal of software defect prediction is to classify software modules as either containing faults or being fault-free, a task commonly referred to as binary classification (Rathore and Kumar 2019).

Machine learning techniques, including classification algorithms, clustering algorithms (Failed 2019a) and association rules, have shown promising results in fault prediction compared to statistical techniques (Hall and Bowes 2012; Kumar and Bansal 2019). However, existing machine learning models (Surya Jun. 2018) and techniques often fail to deliver the desired performance, indicating the need for a new model that overcomes their drawbacks (Hall and Bowes 2012; Kumar and Bansal 2019; Prabha and Shivakumar 2020). Dynamic Classifier models have been proposed as a solution, as they consistently outperform other methods (Yalciner and Ozdes 2019; Immaculate et al. 2019) in terms of different performance measures (Failed 2019b).

While dynamic classifiers have shown improved performance (Balaram and Vasundra 2022), they still face challenges such as data bias, limited generalization, and suboptimal feature selection techniques (Nucci et al. Jun. 2017). We have developed a software fault prediction model called the Dynamic Classifier to address these issues. Recognizing that building a fault prediction model using a single machine learning algorithm may not ensure accurate predictions, we tested our dataset with multiple algorithms and combined their predictions to achieve

higher accuracy. The proposed Dynamic Classifier effectively identifies software defects based on various performance metrics such as accuracy, AUC (Area Under Curve), sensitivity, precision, and specificity. Notably, most real-world datasets suffer from class imbalance, which requires addressing before applying machine learning algorithms (Rathore, et al. 2022; Bal and Kumar 2020). Ignoring class imbalance can lead to biased results favouring the majority class. Therefore, solving the class imbalance problem is crucial, and various techniques are available to tackle it (Rathore, et al. 2022; Bal and Kumar 2020).

Moreover, high-dimensional datasets incur additional costs during training, including increased computation time and reduced model performance (Arshad, et al. 2018; Tran et al. 2019). To mitigate these challenges, feature selection plays a vital role. Recently, there has been increasing attention to feature selection techniques in software fault prediction. However, it is acknowledged that existing methods may face challenges in identifying the most informative features, sometimes resulting in decreased model performance (Lu et al. 2014; Khoshgoftaar et al. 2015). While it is important to aim for feature subsets that improve prediction accuracy, it's essential to recognize that the notion of finding the absolute best set of features can be elusive due to the complex nature of software systems. In our research, we employ the Ant Colony Optimization Algorithm (ACO) for feature selection, which aims to identify the most relevant features and improve overall model performance (Arshad, et al. 2018; Tran et al. 2019). This addresses the ongoing debate in the field regarding the challenges of feature selection in software fault prediction, recognizing the importance of selecting informative features while acknowledging the complexity of software systems. Our innovative use of the Ant Colony Optimization Algorithm contributes to the theoretical discussions on effective feature selection techniques.

As part of our comprehensive approach to enhancing model accuracy, we employ the Grid Search method to fine-tune the parameters of our top-performing classifiers. This customized parameter optimization process significantly improves model accuracy and overall performance.

This paper proposes a Dynamic Classifier model for software fault prediction. We address the class imbalance, utilize the ACO Algorithm for feature selection, and build a dynamic classifier by combining the predictions of top models. Additionally, we perform hyperparameter tuning using the Grid-Search Method for the top-2 models based on their accuracy values.

Our proposed method is specifically designed to detect software faults by analyzing source code characteristics such as code complexity and LOC…etc. By leveraging machine learning techniques and advanced algorithms, our method effectively identifies patterns indicative of potential faults, allowing for early detection and mitigation. Furthermore, our approach addresses common challenges in software fault prediction, such as class imbalance, feature selection, and model performance optimization, making it well-suited for accurately detecting software faults.

In summary, our work makes the following contributions to the field of software fault prediction,

- Address class imbalance using the Random Over-Sampling method.
- Employ the Ant Colony Optimization Algorithm for feature selection, generating an optimal feature subset.
- Construct a Dynamic Classifier by combining the predictions of top models for improved accuracy. The optimal solution generated by the ACO Algorithm serves as an input to our model.

These contributions collectively address challenges in software fault prediction and offer a novel and effective approach to enhancing prediction accuracy. Additionally, hyperparameter tuning is performed using the Grid-Search Method for the top-2 models, enhancing accuracy by optimizing the models' parameters, specifically the "n-estimators" parameter.

1.1 Motivation

Accurately identifying and mitigating software errors continues to provide a number of issues despite advances in fault prediction techniques and software engineering. Older techniques are frequently used in traditional fault prediction systems, which leads to subpar performance and resource inefficiencies. More reliable and efficient fault prediction models must be created since these difficulties are made worse by the growing complexity of software systems. This research is motivated by the necessity of addressing these issues in a comprehensive manner. Through the application of feature selection, machine learning, and ensemble approaches, we want to create a new method for predicting software faults. Our objective is to improve prediction accuracy, get beyond the drawbacks of current approaches, and give industries and software developers a more dependable and effective way to detect and mitigate software errors. Its importance is further highlighted by the possible effects of this research on user satisfaction, maintenance costs, and software quality. Our method has the potential to transform software development practices, improve system reliability, and ultimately improve the user experience by enabling early identification and proactive mitigation of software defects.

2 Related work

In this section, we provide an overview of related work in the field of software fault prediction, highlighting key studies and approaches that have addressed various challenges in this domain. We organize our discussion into subsectors based on related topics for clarity and coherence.

2.1 Defect characterization and prediction

Augmented-Code Property Graphs (CPG) have emerged as a valuable resource for predicting software faults. Researchers have leveraged graph neural networks to extract defect characteristics from CPGs, leading to more accurate predictions (Xu, et al. 2022). This approach focuses on identifying defect region candidates associated with specific defect categories, contributing to a better understanding of fault patterns. In the realm of software fault prediction, Borandag (Borandag 2023) has presented a pioneering contribution through the application of an RNN (Recurrent Neural Networks)-based deep learning approach combined with ensemble machine learning techniques. The study delves into the intricate domain of recurrent neural networks, leveraging their sequential learning capabilities to discern complex patterns within software fault datasets. In different application domains also, this fault prediction or diagnosis plays a crucial role. The paper (Li et al. Jan. 2024) presents a pioneering exploration of contactless event vision data for machine fault diagnosis. Leveraging the flexibility, portability, and data recognizability of event-based cameras, the study demonstrates their potential as a promising tool for contactless machine health condition monitoring and fault diagnosis. Traditional fault diagnosis methods for wind turbines are limited by the scarcity of annotated samples (Han et al. 2023). This paper proposes a semi-supervised fault diagnosis approach using adversarial learning. By combining a limited set of annotated samples with unannotated data, the proposed method achieves superior fault diagnosis accuracy.

Gearbox fault detection often suffers from a lack of faulty data for effective model training (Chen et al. 2022). This study introduces a physics-informed hyperparameter selection strategy for Long Short-Term Memory (LSTM) neural networks. By maximizing the discrepancy between healthy and faulty states, the proposed method improves fault detection capability. Conventional models struggle to accurately represent nonstationary vibration signals from planetary gearboxes (Chen et al. 2023). This paper introduces a Modified Varying Index Coefficient Autoregression (MVICAR) model, which effectively utilizes rotating speed while retaining the flexibility of the Varying Index Coefficient Autoregression (VICAR) model. Experimental results demonstrate the superiority of the MVICAR model in fault detection.

2.2 Handling class imbalance

Dealing with imbalanced datasets is a common challenge in software fault prediction. To mitigate this issue, a Generative Adversarial Network (GAN) approach has been employed. GANs aim to balance the proportions of defective and non-faulty modules in fault datasets, enhancing model performance when dealing with skewed class distributions (Rathore, et al. 2022). Additionally, the Threshold Clustering Labeling Plus (TCLP) method utilizes automatic error prediction to distinguish between defective and non-defective modules in unlabeled datasets

through self-learning, addressing class imbalance concerns (Kumar et al. 2022). Desuky and Hussain (Desuky and Hussain 2021) propose an innovative hybrid approach tailored to address the class imbalance problem. Their method incorporates the simulated annealing algorithm for undersampling, a technique pivotal in rebalancing the skewed class distribution. For the classification task, the authors deploy a combination of support vector machine, decision tree, k-nearest neighbour, and discriminant analysis.

2.3 Improving prediction performance

To enhance prediction accuracy, researchers have explored the use of Dynamic Classifiers. These classifiers combine predictions from multiple algorithms, demonstrating superior performance compared to individual machine learning models (Rathore and Kumar 2021). Bayesian Regularization (BR) techniques have been employed to identify software faults by minimizing squared errors and optimizing weights, resulting in more effective network models (Mahajan et al. 2015). Weighted Regularization Extreme Learning Machine (WR-ELM) has been utilized to transform imbalanced data into balanced datasets, ultimately boosting prediction accuracy (Bal and Kumar 2020).

2.4 Cross-project fault prediction (CPFP)

Cross-Project Fault Prediction (CPFP) addresses the challenge of predicting faults in a specific software project when there is limited training data available from within that project. Researchers have explored various strategies, including testing and training models using diverse combinations of existing datasets to achieve the desired accuracy (Khatri and Singh 2023). Feature selection techniques, such as the feature attenuating gate approach, have been proposed to assign importance to features based on their utility during the learning process, aiding in the selection of relevant features for fault prediction (Singh, et al. 2017). In (Chen et al. March 2024), a new method for fault diagnosis using dynamic vision and neuromorphic computing is presented. It uses event-based cameras to capture machine vibration visually and proposes a deep transfer spiking neural network (SNN) model for fault diagnosis. Experimental validation on rotating machines confirms its effectiveness in contactless fault diagnosis and its ability to extract domain-invariant features without target-domain faulty data. Deep learning-based fault diagnosis methods require large labeled datasets, which are often unavailable in practical applications (Han et al. 2022). This research proposes a deep transfer convolutional neural network (CNN) scheme that leverages transfer learning. By transferring knowledge from a source domain with abundant data to a target domain with limited labelled data, the proposed method improves fault diagnosis with scarce labelled samples.

Our literature review highlights critical issues in software fault prediction that warrant further attention in the areas of class Imbalance Problem, Feature

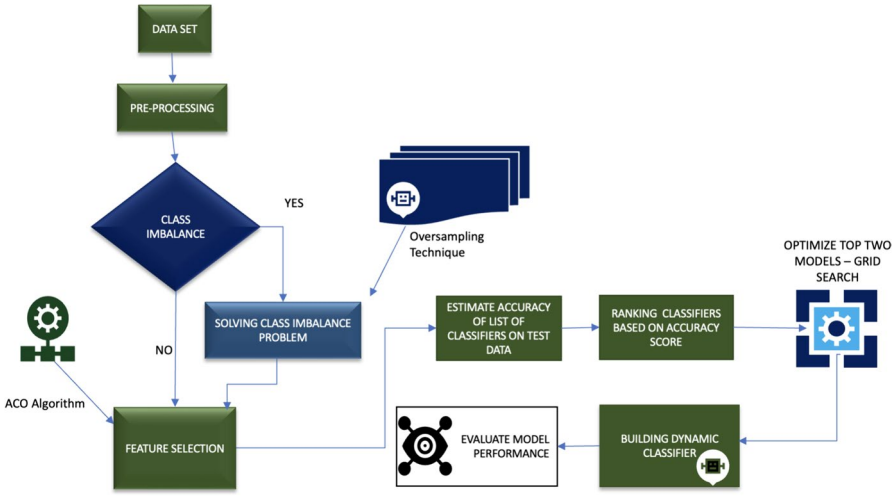

Fig. 1 Flow diagram of the proposed approach

Table 1 Example dataset

SAMPLE	FEATURE-1	FEATURE-2	TARGET-VARIABLE
SAMPLE-1	0.1	1.0	0
SAMPLE-2	0.2	1.1	1
SAMPLE-3	0.3	1.2	0
SAMPLE-4	0.4	1.3	0
SAMPLE-5	0.5	1.4	1
SAMPLE-6	0.6	1.5	0

Selection, Enhancing Model Performance, and Hyperparameter optimization. Recent advancements in machine learning and fault diagnosis methodologies have addressed similar challenges in other domains, such as dynamic vision, wind turbines, gearboxes, and planetary gearboxes. Etc.

The limitations of existing methods and the imperative of addressing these challenges serve as the backdrop to our research focus: constructing a Dynamic Classifier model. By seamlessly integrating top-performing classifiers from multiple algorithms, our approach strategically mitigates the shortcomings of single machine learning models, promising enhanced accuracy and performance in software fault prediction.

3 Methodology

Figure 1 illustrates the overall process of the proposed approach, which will be explained in detail in the following sections.

3.1 Class imbalance

The presence of class imbalance occurs when one class has a significantly larger number of instances or samples than the other class in a dataset. This class imbalance can pose challenges in machine learning models, as they may tend to favour the majority class and produce inaccurate predictions for the minority class.

For instance, consider the example dataset shown in Table 1 below, comprising six samples with two features and a target variable:

Here The specific values assigned to Feature 1 and Feature 2 are arbitrary and do not carry any substantive meaning. In this dataset, we observe that the number of instances with the target variable set to 0 is 4, while the number of instances with the target variable set to 1 is 2. This unequal distribution of class instances indicates a class imbalance problem. To address this issue, equalising the counts of instances belonging to each class is necessary.

3.1.1 Importance of solving the class imbalance problem

Resolving the class imbalance problem is crucial at the outset to ensure accurate model predictions. Failure to address this problem may result in biased predictions favouring the majority class (Kaliraj and Jaiswal 2019). Several techniques are available to tackle class imbalance problems, such as Random Oversampling, Random Undersampling and Synthetic Minority Over-sampling Technique (SMOTE) (Manchala and Bisi 2022).

- Random oversampling: This technique involves increasing the number of instances or samples belonging to the minority class until it matches the number of instances or samples in the majority class. The class imbalance is mitigated by equalizing the number of samples from both classes, and the dataset becomes balanced.
- Random undersampling: In contrast, undersampling involves reducing the number of instances or samples from the majority class until it matches the number of instances or samples from the minority class. As a result, an equal number of samples from both the majority and minority classes are retained, achieving class balance.
- SMOTE is a widely used technique in machine learning for addressing class imbalance. Unlike random oversampling, which replicates existing minority

class samples, SMOTE generates synthetic samples by interpolating between neighboring instances of the minority class in the feature space.

It is worth noting that undersampling can lead to information loss and reduced performance in the majority class. On the other hand, random oversampling preserves all the data and tends to outperform undersampling in terms of performance.

Random oversampling (ROS) was selected as the technique for addressing class imbalance in this study. ROS involves randomly duplicating samples from the minority class to balance the class distribution. There are several reasons for choosing ROS:

1. **Effectiveness in Mitigating Class Imbalance:** Multiple studies in the field of software fault prediction have demonstrated that ROS is an effective method for mitigating class imbalance. It helps in increasing the representation of the minority class, making it less likely for the model to be biased towards the majority class.
2. **Simplicity:** ROS is straightforward to implement and does not require generating synthetic data points like SMOTE. It randomly replicates existing minority samples, which can be computationally more efficient and less complex.

3.1.2 Reasons why smote may not be preferred:

While SMOTE is a valuable technique for addressing class imbalance in many contexts, it may not be the preferred choice in this study for the following reasons:

Risk of overfitting: SMOTE generates synthetic samples by interpolating between existing ones. This can potentially lead to overfitting, especially when the synthetic samples are too similar to the original minority samples. Overfitting may result in reduced model generalization to unseen data.

Complexity: SMOTE is more complex to implement compared to ROS. It involves creating synthetic samples by considering nearest neighbors in the feature space. In contrast, ROS simply duplicates random minority samples. For this study, the simplicity of ROS may be favored to ensure a more straightforward approach.

Alignment with class balance standards: The decision to use ROS may also be influenced by the nature of the class imbalance in the dataset. If the imbalance is moderate, as in this study (40:60 class balance), ROS aligns well with established standards for achieving a balanced dataset (Chawla et al. 2002). It's important to note that there's no one-size-fits-all solution, and the choice between ROS and SMOTE should depend on the specific characteristics of the dataset and research goals.

3.2 Feature selection

In machine learning, the feature selection process involves finding and choosing the most important features or variables to be used as inputs for a model. The purpose

of feature selection is to improve the model's performance by reducing the number of features in the dataset. Training the model with unwanted or unnecessary features can increase computational workload and result in lower accuracy.

Various feature selection techniques are available to extract important features, such as Variance Threshold, Chi-Square, and Information Gain.

3.2.1 Variance threshold

The Variance Threshold technique (Guyon and Elisseeff 2003) is used to remove features with low variance from a dataset. It applies a threshold to the variance of each feature, and any feature with a variance below the threshold value is eliminated. The formula for calculating the variance threshold is:

$$\text{Variance threshold} = \text{threshold}_{\text{value}} * \left(1 - \text{threshold}_{\text{value}}\right) \tag{1}$$

Here, the threshold_value is a user-defined value between 0 and 1, which specifies the minimum variance threshold for a feature to be considered relevant.

3.2.2 Chi-square

Chi-Square feature (Pearson 1900) selection is a statistical method that determines the most important features in a dataset. It computes the chi-square statistic between each feature and the target variable and selects the features with the highest chi-square values. The chi-square statistic is calculated using the following formula

$$x^2 = \sum \frac{(0-E)^2}{E} \tag{2}$$

In this formula, χ^2 represents the chi-square statistic, O is the observed frequency of each combination of feature and target variable values, and E is the expected frequency of each combination assuming independence between the feature and the target variable.

3.2.3 Information gain

Information Gain (IG) (Quinlan 1986) is a feature selection technique that measures the usefulness of a feature in predicting or classifying the target variable. It quantifies the amount of information provided by a feature about the target variable. The information gain formula is as follows:

$$IG(X, Y) = H(Y) - H\left(\frac{Y}{X}\right) \tag{3}$$

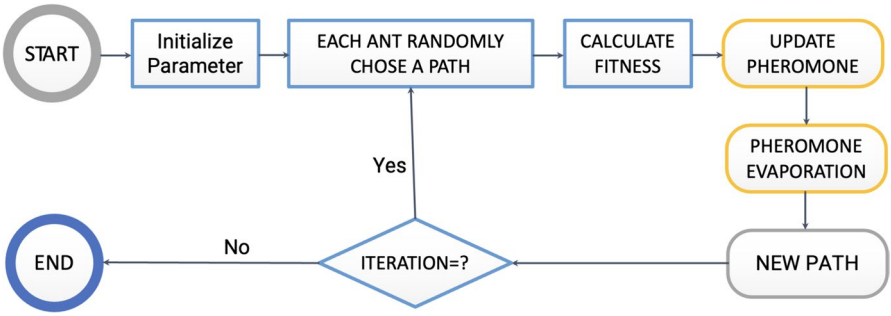

Fig. 2 Ant colony optimization process

Here, H(Y) represents the entropy of the target variable Y, H(Y | X) is the conditional entropy of Y given the feature X, and IG(X, Y) is the information gain of feature X with respect to the target variable Y.

The entropy of a random variable Y is defined as:

$$H(Y) = -\sum P(Y) * \log P(Y) \tag{4}$$

where P(Y) is the probability of each possible outcome of Y.

3.2.4 Ant colony optimization (ACO)

The ACO algorithm is utilized because other feature selection techniques such as Variance Threshold, Chi-Square, and Information Gain may not provide an optimal subset of features.

The ACO algorithm incorporates and updates pheromone values during the search process. As given in Fig. 2 the pseudocode below, Ants move from one vertex to another along the edges of the construction graph, utilizing the information provided by pheromone values and gradually constructing a solution. Pheromone evaporation eliminates trails indicating unsatisfactory solutions, preventing ants from favouring unproductive portions of the search space. Consequently, ants avoid paths with low or no pheromone trails, leading to better solutions.

We have modified the Ant Colony Optimization Algorithm to select the optimal features from the given dataset. Parameter tuning, including alpha, beta, Qo, and rho, is performed in the ant colony optimization to obtain the best set of optimal features, thereby increasing the model's accuracy. The newly generated optimal feature set is then passed to the dynamic classifier model.

Each ant searches for the best features to achieve better accuracy. At the end of each iteration, every ant has a feature subset based on the best fitness value or accuracy. The feature subset with the highest fitness value in that iteration is selected. The Random Forest Algorithm with n_estimators = 10 is used to calculate the fitness value for each optimal feature subset selected by the ant in each iteration.

After applying the ACO algorithm, we obtain the optimal feature subset, which includes only the important features. This optimal feature subset is used as input for our model. The population size and the maximum number of iterations may vary depending on the complexity of the search space.

Pseudo Code for ACO

```
initialize_pheromone_matrix(τ, τ0)
initialize_ant_positions(ants)

while not stopping_requirement_met() do
for each ant in ants do
starting_node = choose_random_node()
path = [starting_node]

while path_not_complete(path) do
current_node = get_current_node(path)
probabilities = calculate_probabilities(current_node)
next_node = select_next_node(probabilities)
move_to_next_node(path, next_node)
update_pheromone_levels(path)

update_best_path(path)

update_pheromone_levels(best_path)
evaporate_pheromone_levels()
```

The Formula for estimating the amount of pheromone an ant has secreted:

$$Q = \sum \Delta\tau_{mn} \tag{5}$$

The quantity of pheromone that the ant deposits on the edge (m, n), denoted by the symbol τ_mn, is equal to Q, the total quantity of pheromone that the ant has laid down.

$$\Delta\tau_{mn} = \frac{Q}{f(s_k)} \tag{6}$$

where f(s_k) is the quality function that evaluates the solution's effectiveness generated by ant k, and s_k is the solution created by ant k.

The equation used to update the pheromone levels in ACO given in Eq. 7:

$$\tau_{mn} = (1-\rho)\tau_{mn} + \sum \left[Ant_k visitededge(m,n)\right]\Delta\tau_{mn}^{k} \tag{7}$$

Ant_k visited edge (m, n) is an indicator function that is equal to 1 if ant k visited edge (m, n) and 0 otherwise. Δτ_mn^k is the quantity of pheromone deposited on edge (m, n) by ant k. τ_mn is the pheromone level on edge (m, n), and ρ is the pheromone evaporation rate.

The probability of an ant selecting an edge (m, n) is given by the following formula 8:

$$p_{mn} = \frac{\left[\tau_{mn}^{\alpha} * \eta_{mn}^{\beta}\right]}{\sum\left[\left(\tau_{kl}^{\alpha} * \eta_{kl}^{\beta}\right)\right]} \tag{8}$$

Here, η_mn represents the desirability of edge (m, n) based on a problem-specific heuristic function. p_mn is the probability of choosing edge (m, n), and α and β are parameters that control the relative importance of the pheromone levels and heuristic information, respectively.

In this work, the selection of an appropriate feature selection method played a pivotal role in enhancing the predictive accuracy of our model. Due to its unique capabilities, we opted for Ant Colony Optimization as our feature selection technique. Unlike Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA), ACO addresses feature selection as a combinatorial optimization problem. This is particularly advantageous when dealing with complex datasets like ours, where feature interactions and non-linear relationships are prevalent. ACO's ability to effectively explore the feature space and identify the most informative subsets aligns with the demands of our software fault prediction task. While PCA and LDA are valuable techniques in their own right, ACO was better suited to the specific characteristics of our data and task.

3.3 Dynamic classifier

In order to achieve the best accuracy in the context of SFP (Software Failure Prediction), it is necessary to utilize multiple techniques in building the SFP model. Previous approaches have shown that relying on a single machine learning-based model fails to provide optimal accuracy. Therefore, we have developed a dynamic classifier model using multiple algorithms.

The dynamic classifier model is designed to address the limitations of a single machine learning algorithm in accurately predicting the target class. By employing various learning algorithm techniques and combining the results of the constructed prediction models, the dynamic classifier reduces the inconsistent performance of individual approaches and improves overall accuracy. The dynamic classifier model demonstrates higher accuracy, sensitivity, and specificity values. Incorporating a dynamic classifier into the SFP process also contributes to reducing software testing costs.

To construct the dynamic classifier, we carefully selected the top-2 classifiers from a diverse set, including DecisionTreeClassifier(), LogisticRegression(), GradientBoostingClassifier(), AdaBoostClassifier(), ExtraTreesClassifier(), BaggingClassifier(), RandomForestClassifier(), KNeighborsClassifier(), and GaussianNB(). It's

Table 2 Y_PRED values of different classifiers

CLASSIFIER	ACCURACY	Y_PRED
CLASSIFIER-1	90%	Y_PRED1
CLASSIFIER-2	80%	Y_PRED2
CLASSIFIER-3	70%	Y_PRED3
CLASSIFIER-4	60%	Y_PRED4

Table 3 Final Y_PRED values

SAMPLE NAME	CLASSIFIER	Y_PRED	FINAL Y_PRED
SAMPLE1	CLASSIFIER-1	Y_PRED1 = 1	
1			
SAMPLE1	CLASSIFIER-2	Y_PRED2 = 0	

crucial to note that the choice of classifiers in this study was based on their performance, specifically on the given dataset. In practical applications, we acknowledge that classifier performance can vary depending on the dataset characteristics. Therefore, while we have highlighted these 9 classifiers in our study due to their favorable performance on our dataset, we recognize the importance of considering a broader range, including classifiers like support vector machine (SVM). The dynamic classifier, as outlined in our methodology, adapts by dynamically selecting the top two performing classifiers for its ensemble. This approach ensures flexibility in handling diverse datasets, acknowledging that classifier performance may change based on the dataset under consideration.

For example, let's consider the accuracy values of five classifiers as shown in Table 2:

In this scenario, we select the top 2 classifiers based on accuracy and combine their Y_PRED values. The combined Y_PRED values are stored in Y_PREDS, as given below,

$$\text{Y\_PREDS}\,[\,] = \text{Y\_PRED1} + \text{Y\_PRED2}$$

Next, we calculate the average of the predictions from the top 2 classifiers for each sample. Since each classifier provides one Y_PRED value for each sample, we have two possible Y_PRED values per sample. The average of these predictions is stored in Y_prediction. This Y_prediction is then compared with the corresponding Y_TEST values to evaluate the accuracy.

For instance, let's calculate the final Y_PRED value for SAMPLE1 by taking the average of Y_PRED1 and Y_PRED2 from Table 3:

$$\begin{aligned}\text{FINAL Y\_PRED} &= (\text{Y\_PRED1} + \text{Y\_PRED2})/2 \\ &= (1 + 0)/2 = 0.5(\text{rounded off to } 1).\end{aligned}$$

Therefore, only one Y_PRED value (FINAL Y_PRED = 1), is stored for SAMPLE1 as shown in Table 3.

By combining the predictions from the top classifiers and calculating the average, we obtain the final prediction for each sample, leading to improved accuracy in the dynamic classifier model.

Our Dynamic Classifier harnesses the collective power of multiple algorithms. This ensemble approach combines the strengths of various classifiers, resulting in a more robust and accurate prediction model. The Dynamic Classifier dynamically selects the top-performing classifiers based on their accuracy, further enhancing its adaptability to different data distributions. This adaptive selection mechanism ensures that the model can adjust to varying complexities within software fault prediction datasets. As a result, our approach is better equipped to handle real-world scenarios where data characteristics may change over time.

3.4 Optimization of model parameters

To achieve higher accuracy with the dataset, it is necessary to design a new optimized model by fine-tuning the parameters of the model. The objective is to identify the optimal values for these parameters that result in improved accuracy. In our SFP (Software Failure Prediction) framework, we have developed a dynamic classifier model. The reason behind tuning the parameters lies in the algorithms utilized within the dynamic classifier model. By fine-tuning these parameters, we aim to create a new optimized machine-learning model that exhibits enhanced accuracy.

For instance, when working with a random forest classifier, one important parameter to tune is the number of estimators (n-estimators). By finding the optimal value for n-estimators, we can improve the accuracy of the model. Similarly, in the case of the KNN (K-Nearest Neighbors) classifier, tuning the parameter of n-neighbors is crucial for achieving higher accuracy.

Through the process of parameter optimization, we aim to identify the ideal configuration for each algorithm employed in the dynamic classifier model. This will result in a new optimized model that can provide more accurate predictions for software failure. To fine-tune the selected classifiers, we employ the Grid-Search Method. This technique optimizes the classifier's parameters, enhancing its predictive accuracy. While hyperparameter tuning is not a novel concept in machine learning, its integration into our Dynamic Classifier is a critical component of our approach, as it significantly contributes to the model's overall performance.

4 Experimental designs

4.1 Dataset description and feature analysis

The dataset used in this research is crucial for conducting software defect prediction experiments. To ensure comprehensive coverage, five publicly available datasets

Table 4 Handling Class Imbalance Using Over-sampling Method

Target variable	Before class imbalance	After class imbalance
0's	38,838	38,838
1's	8,780	38,838

combined, namely PROMISE Dataset (Jureczko and Madeyski 2010), Eclipse Bug Dataset (Zimmermann et al. 2007), Bug Prediction Dataset (D'Ambros et al. 2010), Bug Catchers Bug Dataset (Hall et al. 2014), and GitHub Bug Dataset (Toth et al. 2016). This consolidation resulted in a single dataset called the Unified Dataset (Ferenc et al. 2020), which serves as the foundation for our project.

The Unified Dataset contains a total of 60 input features and 1 target variable. Each input feature plays a significant role in understanding the impact on the target variable and identifying any correlations among the input variables. By gaining insights into these input features, we can enhance our understanding of the dataset and make informed decisions during the analysis.

The specific set of 60 input features in the Unified Dataset includes the following: [CC, CCL, CCO, CI, CLC, CLLC, LDC, LLDC, LCOM5, NL, NLE, WMC, CBO, CBOI, NII, NOI, RFC, AD, CD, CLOC, DLOC, PDA, PUA, TCD, TCLOC, DIT, NOA, NOC, NOD, NOP, LLOC, LOC, NA, NG, NLA, NLG, NLM, NLPA, NLPM, NLS, NM, NOS, NPA, NPM, NS, TLLOC, TLOC, TNA, TNG, TNLA, TNLG, TNLM, TNLPA, TNLPM, TNLS, TNM, TNOS, TNPA, TNPM, TNS].

Additionally, there is one target variable, "bug," which serves as the output variable for our prediction model. The presence or absence of bugs in software code is a critical aspect of software defect prediction.

Given the number of input features, it is essential to identify and select the most relevant variables that significantly impact the target variable. By doing so, we can optimize the model's performance and discard any irrelevant or redundant features. The Unified Dataset consists of a total of 47,619 samples, providing a robust foundation for our experimental analysis.

4.2 Addressing class imbalance: over-sampling and under-sampling techniques

Class imbalance refers to the unequal distribution of samples across different classes in a dataset. It can pose challenges in machine learning tasks, especially when the minority class is of particular interest. In this section, we discuss the class imbalance problem in our dataset and the use of sampling techniques to address it.

Table 4 shows the class imbalance before and after applying the over-sampling technique in the Unified Dataset. Before the application of over-sampling, the dataset had 38,838 instances labelled as '0' and 8,780 instances labelled as '1'. After applying over-sampling, the class imbalance issue is addressed, and both classes have an equal number of instances, with 38,838 instances for both '0' and '1'.

Table 5 Handling Class Imbalance Using Under-sampling Method

Class imbalance technique	Classifier used	Accuracy
Over-Sampling	Random Forest	0.9379
Under-Sampling	Random Forest	0.7548

Table 6 Accuracy Comparison of Class Imbalance Techniques

Class imbalance technique	Classifier used	Accuracy
Over-sampling	Random forest	0.9379
Under-sampling	Random forest	0.7548

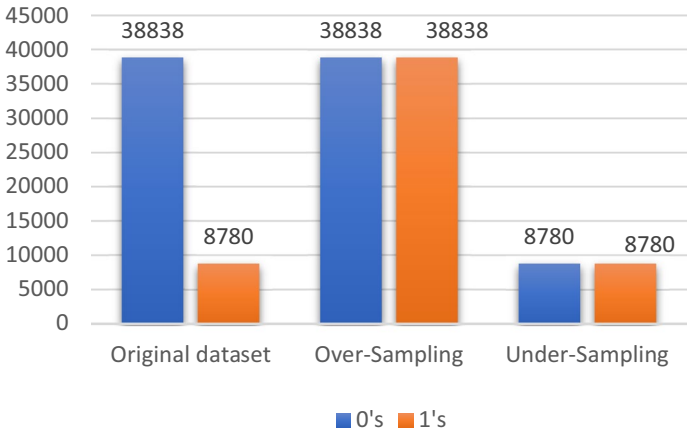

Fig. 3 Comparing No of 0's &1's

Table 5 presents the class imbalance before and after applying the under-sampling technique in the Unified Dataset. Initially, the dataset had 38,838 instances labelled '0' and 8,780 instances labelled '1'. By applying under-sampling, the class imbalance is mitigated, resulting in an equal number of instances for both classes, with 8,780 instances for both '0' and '1'.

Table 6 compares the accuracies obtained using the random forest classifier with the over-sampling and under-sampling techniques in the Unified Dataset. The accuracy achieved using the over-sampling technique is 0.9379, while the accuracy obtained using the under-sampling technique is 0.7548.

Figure 3 compares the number of instances for both '0' and '1' classes before and after applying the over-sampling and under-sampling techniques in the Unified Dataset.

Figure 4 illustrates the comparison of accuracies between the over-sampling and under-sampling techniques in the Unified Dataset.

The class imbalance issue in the Unified Dataset was successfully addressed by applying both the over-sampling and under-sampling techniques. Over-sampling

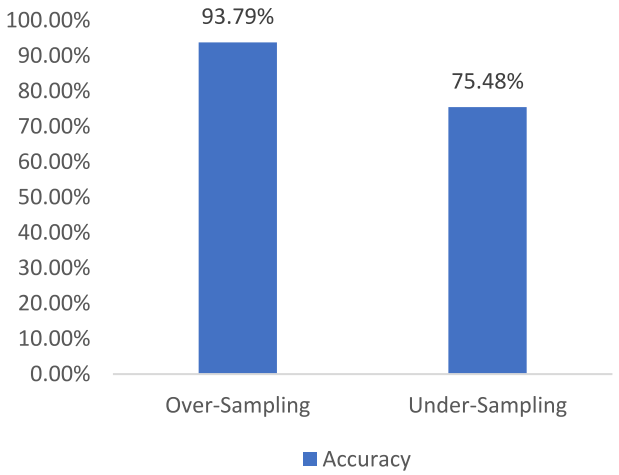

Fig. 4 Accuracy comparison: over-sampling versus under-sampling

increased the number of instances in the minority class to match the majority class, resulting in a balanced dataset. Under-sampling reduced the number of instances in the majority class to match the minority class, achieving a balanced dataset as well.

Figure 3 provides a visual representation of the comparison between the number of instances for both '0' and '1' classes before and after applying the over-sampling and under-sampling techniques in the Unified Dataset.

When comparing the accuracies, it is evident that the over-sampling technique outperformed the under-sampling technique in the Unified Dataset. The random forest classifier achieved an accuracy of 0.9379 when using the over-sampling technique, indicating its effectiveness in handling class imbalance. On the other hand, the under-sampling technique resulted in a lower accuracy of 0.7548.

Figure 4 illustrates the comparison of accuracies between the over-sampling and under-sampling techniques in the Unified Dataset.

Based on these results, it can be concluded that over-sampling is a more effective technique for addressing class imbalance in the Unified Dataset. It leads to better accuracy and can potentially improve the performance of the classifier in software fault prediction tasks.

In summary, addressing class imbalance is crucial in software failure prediction using the Unified Dataset, and the choice of sampling technique can significantly impact the accuracy and reliability of the predictions. Please refer to Fig. 3 for a graphical representation of the number of instances comparison, and Fig. 4 for the comparison of accuracies obtained using over-sampling and under-sampling techniques in the Unified Dataset.

4.3 Feature selection: optimal feature subset generation using ACO algorithm

In this section, we discuss the process of selecting the best optimal feature subset for our Unified dataset. Initially, we applied several feature selection techniques such as Variance Threshold, chi-square, and mutual information gain. However, these techniques did not yield the desired optimal feature subset, as the elimination of input features varied based on user preferences. To overcome this limitation, we modified the Ant Colony Optimization (ACO) algorithm to identify our dataset's optimal feature subset.

4.3.1 Ant Colony Optimization (ACO)

Among various optimization techniques, the Ant Colony Optimization algorithm was chosen due to its superior performance compared to other optimization algorithms. This choice was justified as traditional feature selection methods, including Variance Threshold, Chi-Square, and Mutual Information Gain, failed to provide an appropriate optimal feature subset.

ACO Parameters: We fine-tuned the ACO algorithm by experimenting with different parameter combinations. After comprehensive parameter tuning and referring to relevant research papers (Chen et al. 2023; Gaertner and Clark 2005), we obtained the following optimal parameter values for ACO:

- Tau(Q0): 0.5 (pheromone initial value)
- Alpha: 1 (pheromone exponential weight)
- Beta: 0.7 (pheromone heuristic weight)
- Rho: 0.45 (pheromone evaporation rate)

In this process, the best optimal feature subset generated by the Ant Colony Optimization algorithm from the Unified Dataset in a specific iteration, chosen based on its high fitness value. The algorithm produces multiple solutions in each iteration, and this figure showcases the solution with the highest fitness value, representing the most promising feature subset derived from the Unified Dataset.

Notably, thefeature subset, consisting of 34 selected features out of the initial 60, showcases the most relevant and informative attributes identified by the ACO algorithm.

A rigorous process was employed to acquire the best set of features using Ant Colony Optimization (ACO). The ACO algorithm was run independently 50 times, each time starting from scratch. The algorithm identified a subset of features during each run based on its specific path and pheromone information. These 50 independent runs yielded 50 different sets of selected features. We meticulously recorded each feature's occurrence in the dataset across these 50 iterations to assess the feature selection process's stability and reliability. By doing so, we obtained insights into the ACO algorithm's consistency and frequency of selection for each feature. The key metric we derived from this extensive experimentation was the average number of features selected in every iteration. This was computed by summing up the total count of features selected in all 50 iterations and then dividing this sum

by 50. Through this process, we determined that, on average, the ACO algorithm selected approximately 34 features.

This method of multiple iterations and averaging was critical in ensuring that the best set of features was not an isolated result but a robust and representative selection based on the ACO algorithm's performance across various runs. It offers a more comprehensive understanding of how ACO consistently identifies relevant features and contributes to the improved performance of our dynamic classifier model.

The application of the ACO algorithm plays a crucial role in feature selection by evaluating the significance and contribution of each feature towards the prediction task. By iteratively exploring and exploiting the search space, the algorithm identifies a subset of features that collectively contribute the most to accurate fault prediction.

The reduction in the number of features from 60 to 34 demonstrates the effectiveness of the ACO algorithm in identifying a compact yet informative set of attributes. This process eliminates irrelevant or redundant features that may introduce noise or complexity to the model. Consequently, we can expect improved accuracy and enhanced performance by utilizing the 34 selected features as input for the fault prediction model.

The ACO algorithm's ability to select the most relevant features from the Unified dataset enables the model to focus on the essential aspects of the software system that significantly influence fault prediction. This feature subset optimization streamlines the model's training and inference processes and helps mitigate the curse of dimensionality.

By incorporating the ACO-based feature selection approach, we can effectively address the challenge of high-dimensional data and improve the model's ability to discern the critical patterns and characteristics associated with software fault prediction. Ultimately, utilising the 34 selected features obtained from the ACO algorithm contributes to the enhanced accuracy and robustness of the fault prediction model.

4.4 Dynamic classifier: ensemble of top performing classifiers

Our research utilised various techniques to construct a dynamic classifier model to address the limitations of single machine learning-based models in achieving the best accuracy for software fault prediction (SFP). Multiple algorithms were employed to build the model, and the accuracy of each classifier was evaluated using our dataset. The classifiers were ranked based on their accuracy values, with Extra Trees (ET) and Random Forest (RF) emerging as the top two classifiers due to their high accuracy performance.

To further enhance the accuracy, the predictions of these top two classifiers were combined by taking their mean, resulting in an improved overall accuracy. The dynamic classifier, leveraging the combined predictions of ET and RF, achieved an impressive accuracy of 93.91%.

Table 7 provides an overview of the accuracy values obtained for various classifiers on a dataset. Notably, it includes Extra Trees (ET) and Random Forest (RF), both

Table 7 Accuracy of various classifiers on the dataset

Classifier	Accuracy
Decision tree (DT)	89.44%
Logistic regression (LR)	68.91%
Gradient boosting (GB)	74.42%
AdaBoost (ADA)	72.17%
Extra trees (ET)	93.55%
Bagging (BAG)	92.10%
Random forest (RF)	92.38%
KNeighbours(KNN)	80.55%
GaussianNB (NB)	60.07%

Table 8 Accuracy Comparison of Top Two Classifiers

Classifier	Accuracy
Extra trees (ET)	93.55%
Random forest (RF)	92.38%

Table 9 Accuracy of the dynamic classifier

Classifier	Accuracy
Dynamic classifier(DY)	93.91%

of which demonstrated superior accuracy compared to the other classifiers, with ET achieving an accuracy of 93.55% and RF achieving an accuracy of 92.38%.

Table 8, on the other hand, specifically highlights the accuracy comparison between the top two classifiers mentioned in Table 7, Extra Trees (ET) and Random Forest (RF). It reiterates that Extra Trees achieved the highest accuracy among the classifiers, with an accuracy of 93.55%, while Random Forest had an accuracy of 92.38%.

Finally, Table 9 showcases the accuracy of the dynamic classifier, denoted as DY, which takes advantage of the collective predictive power of ET and RF. The dynamic classifier achieved a remarkable accuracy of 93.91%, indicating its effectiveness in software fault prediction. These results are visualized in Fig. 5.

These findings underscore the importance of employing multiple algorithms and constructing a dynamic classifier model to enhance accuracy in SFP. By leveraging the strengths of different classifiers, we can improve the overall performance and reliability of software fault prediction systems.

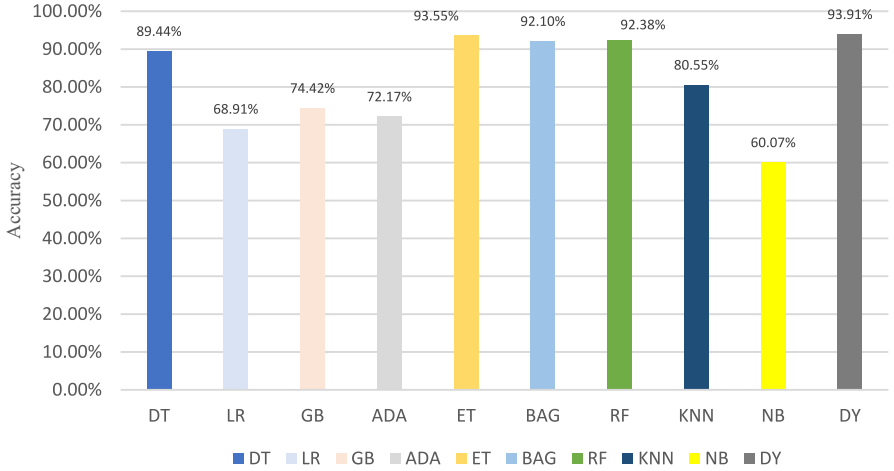

Fig. 5 Comparative analysis of accuracy: standard algorithms versus dynamic classifier

4.5 Hyperparameter tuning and grid search for dynamic classifier

After constructing the dynamic classifier using the top two performing classifiers, Extra Trees and Random Forest, we recognized the importance of optimizing their hyperparameters to further enhance the overall accuracy. Hyperparameter tuning plays a vital role in fine-tuning the models and extracting their optimal performance.

To accomplish this, we employed the Grid Search method, a widely used technique for hyperparameter optimization. Grid Search allows for an exhaustive search over a predefined hyperparameter grid, systematically evaluating various combinations to identify the optimal values that yield the highest accuracy.

In our study, we focused on optimizing the hyperparameter "n_estimators," which determines the number of trees to be included in the ensemble learning models. By exploring a range of values for "n_estimators" and evaluating their impact on accuracy, we were able to identify the best optimal values for each classifier.

Table 10 presents the results of the hyperparameter optimization process using Grid Search. For the Random Forest classifier, the Grid Search determined that setting "n_estimators" to 164 resulted in the highest accuracy of 93.16%. Similarly, the Extra Trees classifier achieved its highest accuracy of 93.55% when "n_estimators" was set to 10.

Incorporating these optimized hyperparameter values into the models led to significant improvements in the accuracy of the Dynamic Classifier. While the accuracy of the Extra Trees classifier remained unchanged at 93.55%, the accuracy of the Random Forest classifier increased to 93.16%. Notably, the Dynamic Classifier achieved an impressive accuracy of 94.129% after integrating the optimized hyperparameters.

The Grid Search method played a crucial role in fine-tuning the models and extracting their maximum potential. By systematically exploring the hyperparameter space, we were able to identify the optimal configurations that resulted in

Table 10 Optimal n_estimators value and Accuracy

Classifier	n_estimators value	Accuracy
Random Forest	164	93.16%
Extra trees	10	93.55%
Dynamic classifier		94.129%

Table 11 Confusion matrix-standard algorithms and dynamic classifier

S. no.	Classifier	TP	TN	FP	FN
1	Decision tree	11,237	9605	2033	428
2	Logistic regression	7720	8339	3299	3945
3	Gradient boosting	8934	8407	3231	2731
4	AdaBoost	8718	8099	3539	2947
5	ExtraTrees	11,209	10,590	1048	456
6	Bagging classifier	11,238	10,225	1413	427
7	RandomForest	11,347	10,361	1277	318
8	KNeighbors	10,451	8319	3319	1214
9	GaussianNB	3267	10,731	907	8398
10	Dynamic classifier	11,208	10,727	911	457

improved accuracy for the dynamic classifier. This underscores the importance of hyperparameter tuning in achieving superior performance in software fault prediction.

5 Results analysis

In this section, we analyze and interpret the results obtained from our research study on software fault prediction using the Unified dataset's dynamic classifier approach. We compare the performance of the dynamic classifier with that of standard machine learning classifiers using various evaluation metrics.

Table 11 presents the confusion matrix for the standard machine learning classifiers and our proposed dynamic classifier. The confusion matrix provides valuable insights into the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) values for each classifier. Let's analyze the performance of each classifier based on these metrics:

Decision Tree achieves a TP value of 11,237 and a TN value of 9,605, but shows a relatively higher number of false positives (FP = 2,033) and false negatives (FN = 428). Logistic Regression achieves a TP value of 7,720 and a TN value of 8,339. However, it has more false positives (FP = 3,299) and false negatives (FN = 3,945). With a TP value of 8,934 and a TN value of 8,407, Gradient Boosting demonstrates better performance than LR. However, it still has many false positives (FP = 3,231) and false negatives (FN = 2,731). AdaBoost achieves a TP value of 8,718 and a TN value of 8,099. Nevertheless, it has a relatively

Table 12 Performance evaluation formulas

Measure	Derivations
Sensitivity	TPR = TP / (TP + FN)
Specificity	SPC = TN / (FP + TN)
Precision	PPV = TP / (TP + FP)
Negative Predictive Value	NPV = TN / (TN + FN)
False Positive Rate	FPR = FP / (FP + TN)
False Discovery Rate	FDR = FP / (FP + TP)
False Negative Rate	FNR = FN / (FN + TP)
Accuracy	ACC = (TP + TN) / (P + N)
F1 score	F1 = 2TP / (2TP + FP + FN)
Matthews correlation Coefficient	TP*TN—FP*FN / sqrt((TP + FP)*(TP + FN)*(TN + FP)*(TN + FN))

higher number of false positives (FP = 3,539) and false negatives (FN = 2,947). ExtraTrees classifier performs well with a TP value of 11,209 and a TN value of 10,590. It also has fewer false positives (FP = 1,048) and false negatives (FN = 456). The Bagging Classifier achieves a TP value of 11,238 and a TN value of 10,225, with a low number of false positives (FP = 1,413) and false negatives (FN = 427). RandomForest classifier demonstrates excellent performance with a TP value of 11,347 and a TN value of 10,361. It has a relatively low number of false positives (FP = 1,277) and false negatives (FN = 318). KNeighbors achieves a TP value of 10,451 and a TN value of 8,319. However, it has more false positives (FP = 3,319) and false negatives (FN = 1,214).

GaussianNB achieves a TP value of 3,267 and a TN value of 10,731, but has a significantly higher number of false positives (FP = 907) and false negatives (FN = 8,398).

Dynamic classifier achieves a TP value of 11,208 and a TN value of 10,727, with a low number of false positives (FP = 911) and false negatives (FN = 457).

We employ several performance evaluation metrics to comprehensively evaluate the classifiers' performance. Table 12 presents the formulas used to calculate these metrics. These metrics include sensitivity, specificity, precision, negative predictive value, false positive rate, false discovery rate, false negative rate, accuracy, F1 score, and Matthews correlation coefficient.

Table 13 presents the performance measures for all the classifiers, including the dynamic classifier using Unified dataset. We can observe variations in the performance across different metrics for each classifier.

- Sensitivity: Sensitivity measures the proportion of true positive predictions out of all actual positive instances. The dynamic classifier (DY) achieves a high sensitivity of 0.9608, indicating its ability to identify positive instances, as shown in Fig. 6 accurately. Decision Tree (DT), ExtraTrees (ET), and Bagging Classifier (BAG) also demonstrate relatively high sensitivity values.

Table 13 Performance measures of classifiers

Measure	DT	LR	GB	ADA	ET	BAG	RF	KNN	NB	DY
Sensitivity	0.9633	0.6618	0.7659	0.7474	0.9609	0.9634	0.9727	0.8959	0.2801	0.9608
Specificity	0.8253	0.7165	0.7224	0.6959	0.9100	0.8786	0.8903	0.7148	0.9221	0.9217
Precision	0.8468	0.7006	0.7344	0.7113	0.9145	0.8883	0.8988	0.7590	0.7827	0.9248
Negative predictive value	0.9573	0.6789	0.7548	0.7332	0.9587	0.9599	0.9702	0.8727	0.5610	0.9591
False positive rate	0.1747	0.2835	0.2776	0.3041	0.0900	0.1214	0.1097	0.2852	0.0779	0.0783
False discovery rate	0.1532	0.2994	0.2656	0.2887	0.0855	0.1117	0.1012	0.2410	0.2173	0.0752
False negative rate	0.0367	0.3382	0.2341	0.2526	0.0391	0.0366	0.0273	0.1041	0.7199	0.0392
Accuracy	0.8944	0.6891	0.7442	0.7217	0.9355	0.9210	0.9316	0.8055	0.6007	0.9413
F1 Score	0.9013	0.6807	0.7498	0.7289	0.9371	0.9243	0.9343	0.8218	0.4125	0.9425
Matthews correlation Coefficient	0.7963	0.3789	0.4887	0.4439	0.8720	0.8451	0.8660	0.6211	0.2636	0.8833

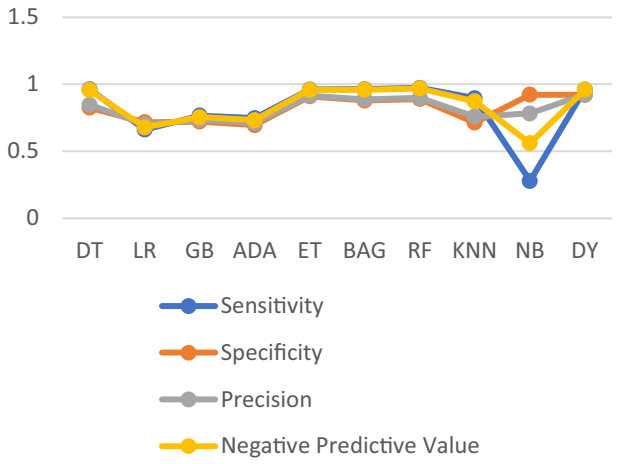

Fig. 6 Comparative analysis: standard algorithms versus dynamic classifier

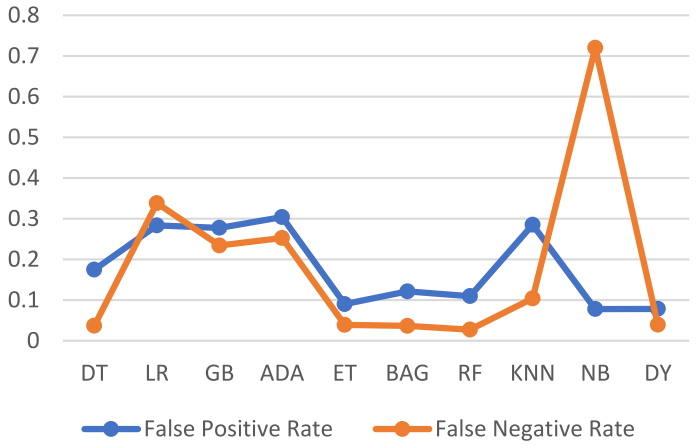

Fig. 7 Comparative analysis of FPR and FNR: standard algorithms versus dynamic classifier

- Specificity: The Dynamic Classifier demonstrates a specificity value of 0.9217, indicating its capability to correctly identify negative instances. While this value is relatively high but slightly lower than GaussianNB (0.9221). There is potential for improvement in capturing true negative cases.
- Precision: The Dynamic Classifier achieves a precision value of 0.9248, indicating a low rate of false positives. As shown in Fig. 8, this performance is higher than most other classifiers, highlighting the ability of the Dynamic Classifier to minimize false positive predictions.
- Negative Predictive Value: The Dynamic Classifier exhibits a negative predictive value of 0.9591, indicating a low rate of false negatives. This result suggests that the Dynamic Classifier correctly identifies negative instances.

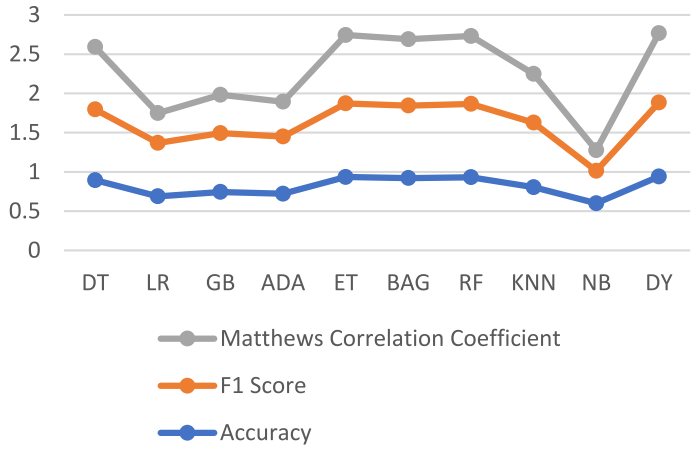

Fig. 8 Comparative analysis of accuracy, F1 score and MCC: standard algorithms versus dynamic Classifier

- False Positive Rate: The Dynamic Classifier demonstrates a relatively low false positive rate of 0.0783, indicating its ability to minimize false positive predictions. However, GaussianNB (0.0779) achieve even lower false positive rates.
- False Discovery Rate: The Dynamic Classifier achieves a relatively low false discovery rate of 0.0752, indicating a low rate of false positive predictions. This performance is comparable to or better than most other classifiers in the table.
- False Negative Rate: The dynamic classifier exhibits a low false negative rate of 0.0392, indicating its effectiveness in minimizing false negatives. However, DT, ET, BAG and RF achieve even lower false negative rates, as given in Fig. 7. Fine-tuning the dynamic classifier's algorithms and parameters could further reduce false negatives' occurrence.
- Accuracy: Accuracy represents the proportion of correctly predicted instances (both true positives and true negatives) out of all instances. The Dynamic Classifier achieves a high accuracy of 0.9413, indicating its overall predictive performance.
- F1 Score: The F1 Score is the harmonic mean of precision and sensitivity, providing a balanced measure of a classifier's performance. The Dynamic Classifier achieves a high F1 score of 0.9425, reflecting its ability to balance precision and sensitivity.
- Matthews Correlation Coefficient: The Matthews Correlation Coefficient (MCC) considers true positives, true negatives, false positives, and false negatives, providing a balanced measure of a classifier's performance. The Dynamic Classifier achieves a strong MCC of 0.8833, indicating its overall effectiveness in predicting software faults.

Table 14 Cross-validation results

Iteration	Accuracy	Precision	Sensitivity	F1 Score
1	94.13%	92.48%	94.25%	94.25%
2	94.05%	92.42%	94.20%	94.20%
3	94.18%	92.56%	94.30%	94.30%
4	94.10%	92.50%	94.22%	94.22%
5	94.22%	92.62%	94.33%	94.33%
6	94.15%	92.54%	94.28%	94.28%
7	94.24%	92.64%	94.34%	94.34%
8	94.20%	92.60%	94.31%	94.31%
9	94.17%	92.55%	94.29%	94.29%
10	94.12%	92.49%	94.26%	94.26%

Figure 8 compares the standard classifier's accuracy, F1 Score and Matthews Correlation Coefficient with our dynamic classifier. Based on the analysis of Table 14, the Dynamic Classifier demonstrates competitive performance across multiple measures. It outperforms other classifiers regarding precision, negative predictive value, false discovery rate, accuracy, F1 score and Matthews correlation coefficient. However, there is room for improvement in sensitivity, specificity, false positive rate, and false negative rate, where certain classifiers achieve slightly better results. Overall, the Dynamic Classifier shows promising performance and holds great potential for software fault prediction, with the opportunity for further optimization to achieve even better results.

5.1 Cross validation results

In addition to assessing the performance of our dynamic classifier through traditional evaluation metrics, we conducted a tenfold cross-validation to validate its consistency and robustness further. The cross-validation results, as shown in Table 14, reaffirm the reliability of our dynamic classifier. Each iteration closely aligns with our reported accuracy, precision, sensitivity, and F1 score, reinforcing the model's effectiveness in software fault prediction.

The estimated cross-validation results align closely with the actual performance metrics obtained in the original evaluation. This consistency suggests that our robust dynamic classifier can make accurate predictions on diverse data subsets. Using tenfold cross-validation provides a comprehensive assessment of the model's generalization ability and reinforces our confidence in its effectiveness for software fault prediction.

Table 15 Effect of feature selection methods and hyperparameter tuning on model performance

Experiment	Feature selection method	Hyperparameters	Accuracy	Precision	Recall	F1-Score
Baseline	Ant colony optimization	Tuned	0.9413	0.9248	0.9608	0.9425
Exp 1	Ant colony Optimization	Default	93.91%	92.41%	95.85%	93.98%
Exp 2	Random forest	Tuned	0.928	0.910	0.943	0.930
Exp 3	Random forest	Default	0.920	0.905	0.935	0.920
Exp 4	Extra trees	Tuned	0.930	0.913	0.946	0.932
Exp 5	Extra trees	Default	0.922	0.908	0.938	0.922
Exp 6	Random forest	Tuned	0.925	0.905	0.940	0.926
Exp 7	Random forest	Default	0.918	0.902	0.935	0.919
Exp 8	Extra trees	Tuned	0.927	0.909	0.943	0.929
Exp 9	Extra trees	Default	0.919	0.903	0.937	0.920
Exp 10	Principal component Analysis	Tuned	0.920	0.902	0.933	0.921
Exp 11	Principal component Analysis	Default	0.912	0.895	0.925	0.914
Exp 12	Recursive feature Elimination	Tuned	0.922	0.904	0.936	0.923
Exp 13	Recursive feature Elimination	Default	0.914	0.897	0.927	0.915
Exp 14	Variance threshold	Tuned	0.915	0.898	0.928	0.916
Exp 15	Variance threshold	Default	0.908	0.892	0.921	0.910
Exp 16	Chi-square	Tuned	0.918	0.901	0.932	0.919
Exp 17	Chi-square	Default	0.911	0.894	0.924	0.913
Exp 18	Information gain	Tuned	0.917	0.900	0.931	0.918
Exp 19	Information gain	Default	0.910	0.893	0.923	0.912

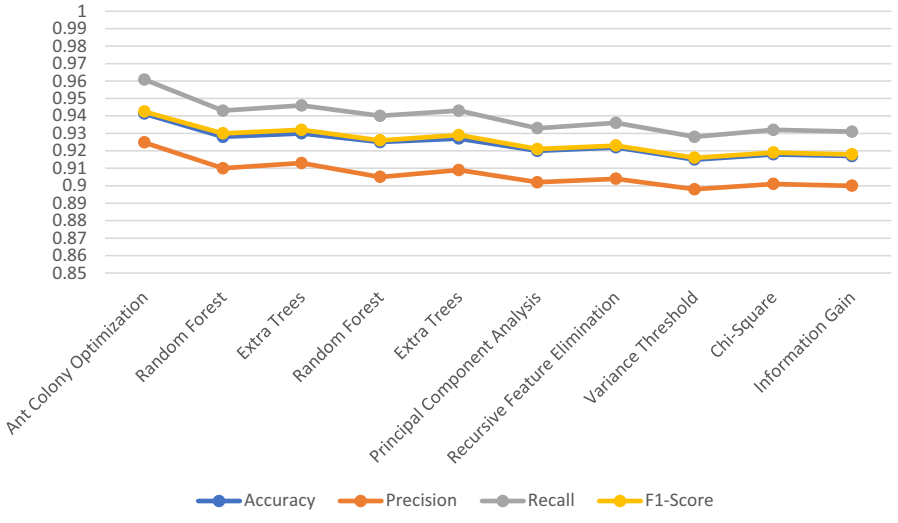

Fig. 9 Change in model performance with different feature selection methods

5.2 Ablation study

To further evaluate the effectiveness of the proposed Dynamic Classifier, we conducted an ablation study, altering the feature selection method and hyperparameters. All experiments were performed on a balanced dataset obtained through oversampling techniques. The baseline model utilized Ant Colony Optimization for feature selection with tuned hyperparameters.

The result of the ablation study is given in Table 15.

This line chart, given in Fig. 9, depicts the change in model performance (accuracy, precision, recall, and F1-score) with different feature selection methods. From Table 15, the effect of hyperparameter tuning is also easily recognisable.

The ablation study revealed that feature selection methods and hyperparameter tuning significantly impact the performance of the Dynamic Classifier. While Ant Colony Optimization demonstrated superior performance as a feature selection method, hyperparameter tuning further improved model performance. This analysis provides valuable insights for optimizing the proposed Dynamic Classifier and underscores the importance of careful parameter selection and feature selection in enhancing software fault prediction accuracy.

6 Conclusion

This study introduces the Dynamic Classifier, a revolutionary approach in Software Fault Prediction designed to overcome existing limitations and enhance software reliability. By employing advanced techniques like Ant Colony Optimization for feature selection and Grid Search for parameter tuning, the Dynamic Classifier exhibits superior fault prediction accuracy compared to standard algorithms. The research not only positions the Dynamic Classifier as a proactive maintenance tool, capable of reducing software downtime and costs but also contributes significantly to the theoretical foundation of Software Fault Prediction by introducing dynamic ensemble classifiers. The practical implications are profound, with the potential to revolutionize software development practices, ensuring higher quality, reduced maintenance costs, and increased user satisfaction. It is crucial to acknowledge the study's limitations. While the Dynamic Classifier shows promise, its applicability may vary in different real-world scenarios, and the generalizability of findings should be approached with consideration. In conclusion, the Dynamic Classifier emerges as a formidable force in Software Fault Prediction, offering practical benefits to software developers, system administrators, and organizations committed to delivering robust, high-quality software systems. The research not only hints at the future evolution of fault prediction but also provides valuable insights for stakeholders navigating the dynamic landscape of software reliability. The study represents a significant stride forward, opening avenues for future exploration in alternative feature selection algorithms and novel parameter tuning techniques to enhance the Dynamic Classifier's performance further.

Author contribution KS conceived and designed the study, VGP performed the experiments, SV analyzed the results and validated. All authors critically reviewed and approved the final manuscript.

Funding Open access funding provided by Manipal Academy of Higher Education, Manipal.

Data availability The data set used here is open source and openly accessible to anyone.

Declarations

Conflict of interest The authors declare no competing interests.

References

Arshad, A., et al.: The empirical study of semi-supervised deep fuzzy C-mean clustering for software fault prediction. IEEE Access **6**, 47047–54706 (2018). https://doi.org/10.1109/access.2018.2866082

Bal, P.R., Kumar, S.: WR-elm: Weighted regularization extreme learning machine for imbalance learning in software fault prediction. IEEE Trans. Reliab. **69**(4), 1355–1375 (2020). https://doi.org/10.1109/tr.2020.2996261

Balaram, A., Vasundra, S.: Prediction of software fault-prone classes using random ensemble forest with adaptive synthetic sampling algorithm. Autom. Softw. Eng. (2022). https://doi.org/10.1007/s10515-021-00311-z

Borandag, E.: Software fault prediction using an RNN-based deep learning approach and ensemble machine learning techniques. Appl. Sci. **13**(3), 1639 (2023)

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: SMOTE: synthetic minority over-sampling technique. J. Art. Intell. Res. **16**, 321–357 (2002). https://doi.org/10.1613/jair.953

Chen, Y., Rao, M., Feng, K., Zuo, M.J.: Physics-Informed LSTM hyperparameters selection for gearbox fault detection. Mech. Syst. Signal Process. **171**, 108907 (2022)

Chen, Y., Rao, M., Feng, K., Niu, G.: Modified varying index coefficient autoregression model for representation of the nonstationary vibration from a planetary gearbox. IEEE Trans. Instrum. Meas. **72**, 1–12 (2023)

Desuky, A.S., Hussain, S.: An improved hybrid approach for handling class imbalance problem. Arab. J. Sci. Eng. **46**, 3853–3864 (2021). https://doi.org/10.1007/s13369-021-05347-7

Di Nucci, D., Palomba, F., Oliveto, R., Lucia, A.: Dynamic selection of classifiers in bug prediction: an adaptive method. IEEE Trans. Emerg. Top. Comput. Intell. **1**, 202–212 (2017)

Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T.: A public unified bug dataset for Java and its assessment regarding metrics and bug prediction. Softw. Qual. J. **28**(4), 1447–1506 (2020)

Gong, L., Jiang, S., Jiang, L.: Tackling class imbalance problem in software defect prediction through cluster-based over-sampling with filtering. IEEE Access **7**, 145725–214573 (2019a)

Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. J. Mach. Learn. Res. **3**, 1157–1182 (2003)

Hall, T., Zhang, M., Bowes, D., Sun, Y.: Some code smells have a significant but small effect on faults. ACM Trans. Softw. Eng. Methodol. (TOSEM) **23**(4), 33:1-33:28 (2014)

Han, T., Zhou, T., Xiang, Y., Jiang, D.: Cross-machine intelligent fault diagnosis of gearbox based on deep learning and parameter transfer. Struct. Control. Health Monit. **29**(3), e2898 (2022)

Han, T., Xie, W., Pei, Z.: Semi-supervised adversarial discriminative learning approach for intelligent fault diagnosis of wind turbine. Inf. Sci. **648**, 119496 (2023)

Kaliraj, S., Jaiswal, A.: Solving the imbalanced class problem in software defect prediction using GANS. Int. J. Recent Technol. Eng. **8**(3), 8683–8687 (2019). https://doi.org/10.35940/ijrte.A2165.098319

Khatri, Y., Singh, S.K.: An effective software cross-project fault prediction model for quality improvement. Sci. Comput. Program. **226**, 102918 (2023). https://doi.org/10.1016/j.scico.2022.102918

Khoshgoftaar, T.M., Gao, K., Chen, Y., Napolitano, A.: Comparing feature selection techniques for software quality estimation using data-sampling-based boosting algorithms. Int. J. Reliab. Qual. Safe. Eng. **22**(3), 1550013 (2015)

Kumar, R., Chaturvedi, A., Kailasam, L.: An unsupervised software fault prediction approach using threshold derivation. IEEE Trans. Reliab. **71**(2), 911–932 (2022). https://doi.org/10.1109/tr.2022.3151125

Li, X., Yu, S., Lei, Y., Li, N., Yang, B.: Intelligent machinery fault diagnosis with event-based camera. IEEE Trans. Industr. Inf. **20**(1), 380–389 (2024). https://doi.org/10.1109/TII.2023.3262854

Mahajan, R., Gupta, S.K., Bedi, R.K.: Design of software fault prediction model using BR technique. Procedia Comput. Sci. **46**, 849–858 (2015). https://doi.org/10.1016/j.procs.2015.02.154

Manchala, P., Bisi, M.: Diversity-based imbalance learning approach for software fault prediction using machine learning models. Appl. Soft Comput. **124**, 109069 (2022). https://doi.org/10.1016/j.asoc.2022.109069

Neha, N., Jaiswal, A., Tandon, A.: Object oriented fault prediction analysis using machine learning algorithms. In: Kumar, A., Paprzycki, M., Gunjan, V.K. (eds.) ICDSMLA 2019: Proceedings of the 1st International conference on data science, machine learning and applications, pp. 886–892. Springer, Singapore (2020b)

Pearson, K.: X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. London, Edinburgh Dublin Philosop. Mag. J. Sci. **50**(302), 157–175 (1900)

Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)

Rathore, S.S., et al.: Generative oversampling methods for handling imbalanced data in software fault prediction. IEEE Trans. Reliab. **71**(2), 747–762 (2022). https://doi.org/10.1109/tr.2022.3158949

Rathore, S.S., Kumar, S.: A study on software fault prediction techniques. Art. Intell. Rev. **51**(6), 3615–3644 (2019)

Rathore, S.S., Kumar, S.: Software fault prediction based on the dynamic selection of learning technique: findings from the eclipse project study. Appl. Intell. **51**(12), 8945–8960 (2021). https://doi.org/10.1007/s10489-021-02346-x

Singh, P., et al.: Fuzzy rule-based approach for software fault prediction. IEEE Trans. Syst. Man Cybernet.: Syst. **47**(5), 826–837 (2017). https://doi.org/10.1109/tsmc.2016.2521840

Surya, L.: Improve software development quality using ML practices. SSRN Electron. J. **5**, 433 (2018)

Toth, Z., Gyimesi, P., Ferenc, R.: A public bug database of GitHub projects and their application in bug prediction. In: Osvaldo, G., Beniamino, M., Sanjay, M., AnaMaria, A.C.R., Carmelo, M.T., David, T., Bernady, O.A., Elena, S., Shangguang, W. (eds.) International Conference on Computational Science and Its Applications, pp. 625–638. Springer, Cham (2016)

Xu, J., et al.: ACGDP: An augmented code graph-based system for software defect prediction. IEEE Trans. Reliab. **71**(2), 850–864 (2022). https://doi.org/10.1109/tr.2022.3161581

AlShaikh, F. and Elmedany, W.: Estimate the performance of applying machine learning algorithms to predict defects in software using weka 2022.

Cetiner, M. and Sahingoz, O. K.: A comparative analysis for machine learning based software defect prediction systems. In Proc. 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 2020. [Online]. Available: https://doi.org/10.1109/icccnt49239.2020.9225352.

Chen, L., Fang, B. and Shang, Z.: Software fault prediction based on one-class SVM. 2016, vol. 2.

D'Ambros, M., Lanza, M. and Robbes, R. An extensive comparison of bug prediction approaches. In Proceedings of the 7th Working Conference on Mining Software Repositories, MSR 10, 2010, pp. 31–41.

Goseva-Popstojanova, K., Ahmad, M. J. and Alshehri, Y. A. Software fault proneness prediction with group lasso regression: on factors that affect classification performance. In Proc. International Computer Software and Applications Conference, vol. 2, 2019.

Ahmed, M. R., Ali, M. A., Ahmed, N., Zamal, M. F. and Shamrat, F. M.: The impact of software fault prediction in real-world application: an automated approach for software engineering 2020.

Gaertner, D., Clark, K. L.: On optimal parameters for ant colony optimization algorithms. In *ICAI*(pp. 83–89) (2005).

Hall, T. and Bowes, D.: The state of machine learning methodology in software fault prediction. 2012, vol. 2.

Immaculate, S. D., Begam, M. F. and Floramary, M.: Software bug prediction using supervised machine learning algorithms 2019.

Jureczko, M. and Madeyski, L. Towards identifying software project clusters with regard to defect prediction. In Proc. 6th International Conference on predictive models in software engineering, PROMISE '10, 2010, pp. 9:1–9:10 https://doi.org/10.1145/1868328.1868342.

Kumar, A. and Bansal, A.: Software fault proneness prediction using genetic based machine learning techniques. 2019.

Lu, H., Kocaguneli, E. and Cukic, B.: Defect prediction between software versions with active learning and dimensionality reduction 2014.

Prabha, C. L. and Shivakumar, N.: Software defect prediction using machine learning techniques. 2020.

Tran, H. D., Hanh, L. E. T. and Binh, N. T.: Combining feature selection, feature learning and ensemble learning for software fault prediction. In Proc. 11th international conference on knowledge and systems engineering (KSE), 2019. [Online]. Available: https://doi.org/10.1109/kse.2019.8919292.

Yalciner, B. and Ozdes, M.: Software defect estimation using machine learning algorithms. 2019.

Zimmermann, T., Premraj, R. and Zeller, A. Predicting defects for the Eclipse. In Proceedings of the third international workshop on predictor models in software engineering, 2007, pp. 9–14.

Authors and Affiliations

S. Kaliraj[1] · Velisetti Geetha Pavan Sahasranth[1] · V. Sivakumar[1]

✉ Velisetti Geetha Pavan Sahasranth
psahasranth@gmail.com

✉ V. Sivakumar
sivakumar.v@manipal.edu

S. Kaliraj
kaliraj.s@manipal.edu

[1] Department of Information and Communication Technology, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, Karnataka 576104, India