International Journal of Multidisciplinary and Applied Studies (IJMDAS)
E-ISSN:3107-8028

Research Article
Vol. 1, No.3, September 2025

AI-Powered Software Quality Assessment and Defect Prediction for Healthcare IoT Devices

Javid Syed
Software Quality Engineer,USA. Javidsyed9999@gmail.com

ABSTRACT:

It is very important for software defect prediction to improve software quality, reduce cost of maintenance and identify the modules which are prone to software defect in software development process. Traditional classification techniques normally confront the problems of modelling complicated interactions between software metrics and detecting malfunctioning modules correctly when dealing with imbalanced information. The approach uses two benchmark datasets, KC1 and PC1, which are available from the NASA PROMISE repository, and contain software quality attributes, such as code size, cyclomatic complexity, Halstead metrics and control-flow characteristics. The preprocessing pipeline consists of data cleansing, missing value handling, duplication removal, feature scaling, and class imbalance control to boost the reliability of classification. The ML methods included are Bagging, RF, MLP, Radial Basis Function, NB, MNB, SVM, XGBoost and CatBoost. LIME and SHAP are used to make predictions that are interpretable and FastAPI is used to deploy the model. The accuracy, precision, recall, F1-score, ROC-AUC, PR-AUC and Matthews Correlation Coefficient are used to measure performance. According to provisional estimations, CatBoost has the highest accuracy (89.7 % on KC1 and 95.8 % on PC1) which is consistent with what was expected of it. The proposed method combines the complex ensemble classification, explainable AI and API deployment that facilitates robust software fault estimation.

Keywords: Software defect prediction, Machine learning, Ensemble learning, XGBoost, Explainable artificial intelligence, Software quality estimation."

Received Date: 5 September 2025; **Accepted Date:** 15 September 2025; **Published Date:** 20 September 2025

This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author(s) and the source are properly cited.

1. Introduction

Modern applications are written, deployed and maintained keeping software quality in mind. Among the many concerns developers and QA departments have, dependability has grown more significant due to the growing complexity of software systems. Defects may lead to failures, loss of functionality, increased maintenance cost and decreased user satisfaction. Finding defect-prone modules early on makes it easier for teams to prioritise the testing and make the best use of their resources. Hence, it is vital for quality management to consider software defect prediction. Software metrics are used to characterise

Paper Available at: https://ijmdas.org
DOI: https://doi.org/10.65477/ijmdas.2025.v1.i3.02

the code complexity, size and structure and provide information to identify potential flaws before the delivery [1]. Using data to make predictions can aid in defect identification and selection for maintenance [2].

It is still difficult to find out faulty modules precisely. Manual inspection and rule-based evaluation is dependent on the expertise of the inspector and the requirements set out in the rules, which are not useful for complex systems. The class distribution of datasets of defects is usually unbalanced, with the defective classes being the minority. This may lead the classifiers to favour the non-defective modules . But the variety of software environments makes it hard to discover regularities in software faults. The models in current approaches might not capture the nonlinear relationships between software metrics and might not work well on datasets. Reliability of classification, dealing with class imbalance, and comprehensive evaluation are the issues that have been studied in past works [3]. The variables that influence automated forecasts are not easily interpretable, making it difficult to understand the forecasts [4]. These problems require an effective and meaningful estimation of defects.

The aim is to identify possibly buggy software modules based on publicly available benchmark datasets. The methodology examines categorisation algorithms on data sets of different attributes and defect distribution. Special attention is given to the consistency of the forecast and to explaining predictions and features contributions. It also plans to provide trained models for real-world applications. It is characterized by its contributions in the areas of comparative evaluation, better predictions, XAI for transparency, and deployment assistance via an application interface [5]. This helps with systematic quality evaluation and informed defect management [6].

The framework can aid the proactive quality assurance and minimize effort required to discover defect prone modules. These more accurate predictions can help developers focus their inspection, testing and maintenance efforts more efficiently. Interpretable outputs may enhance the trust for automatic categorisation by unveiling the characteristics of the software that led to problems. An application programming interface can be incorporated into software development and quality assurance environment. A comparison of the defect estimation methods across benchmark data sets may provide some information for consistency and utility. The framework, as a whole, is designed to enable an accurate assessment of software defects and hence improve software maintenance decisions and software quality [7]. It is based on the predicted performance and interpretability that it can drive intelligent quality management [8]. They allow for software systems to be dependable in practice [9] and for defect prioritisation and continuous improvements [10].

2. Related Work

The problem of software defect prediction has been of interest because it can be useful in improving software reliability and effective quality assurance. Recognizing the importance of imbalanced class distribution to detect problematic modules, Wang and Yao [11] investigated class imbalance learning problem for software defect prediction. They found that traditional classification methods may give biased predictions if defective occurrences were under-represented. Hussain et al. [12] evaluated ensemble learning methods for software defect prediction and investigated the efficiency of the ensemble learning methods for improving the classification performance. Their research indicated that using multiple prediction models to achieve more accurate predictions than using a single prediction model. The use of ensemble approaches is, however, dependent on the properties of the dataset and the model design, so a detailed comparison is needed.

Automated model selection and optimisation are also great approaches in software fault prediction. To automate model generation, Feurer et al. [13] put forth an efficient and resilient ML system that simultaneously selects methods and optimizes hyperparameters. Their contribution shows how automated methodologies can be used to find optimal predictive setups, thus decreasing the amount of manual experimentation required. Tanaka and Monden [14] assessed the capability of auto-sklearn

Paper Available at: https://ijmdas.org
DOI: https://doi.org/10.65477/ijmdas.2025.v1.i3.02

to be used for software bug prediction and the efficiency of selecting the appropriate categorisation settings. Tanaka et al. [15] have taken a step further to investigate the potential of automatic ML for software bug prediction and emphasized its ability to enhance the selection of the predictive model. Although these methods provide helpful automation features, they are only effective when applied to datasets provided that have certain properties and are also evaluated with certain verification methods that are reliable.

Comparative model assessment and effort-aware prediction are other trends in software quality estimate. Kamei et al. [16] re-evaluated the results of bug prediction, which were usually presented using effort-aware evaluation models, and highlighted the need of accounting the testing effort to evaluate the effectiveness of the defect prediction. The authors' findings indicate that the effectiveness of the defect prediction methods might not be fully captured using traditional evaluation measures alone. Lessmann et al. [17] provided a benchmarking methodology for the comparison of classification models to be used in software defect prediction, and provided empirical insights comparing the relative performance of several classifiers. Their work provided a basis for a standardised examination and systematic comparison of models. However, the different dataset distributions, evaluation criteria and experimental setups still bring obstacles to evaluate the overall performance of the prediction algorithms on numerous software repositories .

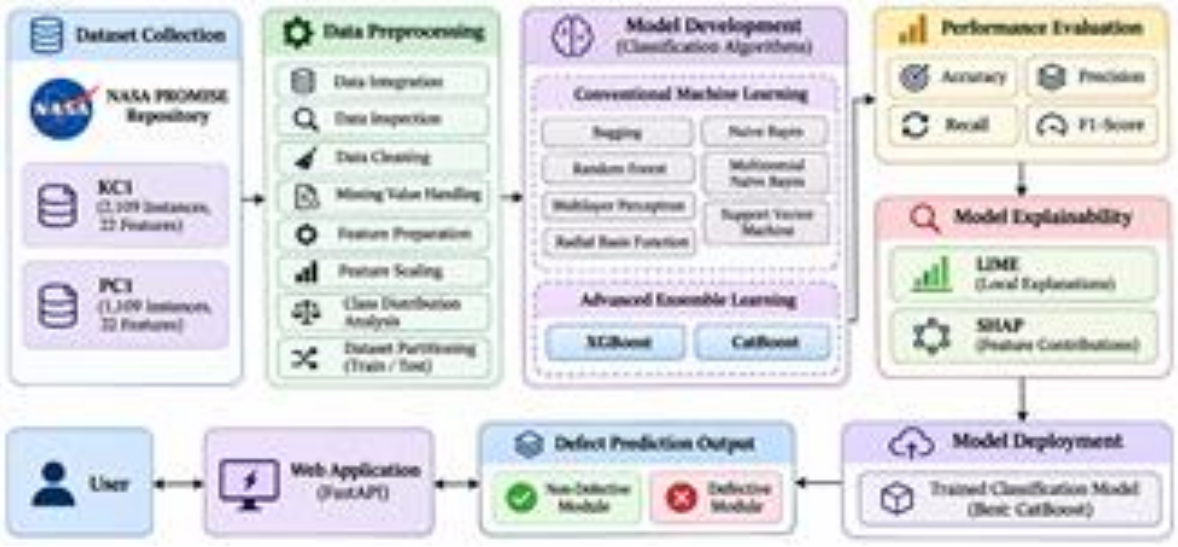
The study of dealing with class imbalance and improving model optimisation has also been extensively researched. To solve class imbalance problems, Bennin et al. [18] proposed a method named as diversity based oversampling called MAHAKIL. The proposed approach indicated the importance of representative sampling of minority classes for improving defect detection. Fu et al. [19] suggest that instead of grid search, DE could be used to tune defect predictors and they suggest potential benefits of using evolutionary optimisation for obtaining a successful configuration. Monden et al. [20] investigated a heuristic rule reduction strategy to software fault-proneness prediction. The authors demonstrated the usefulness of keeping the prediction rules simple but still retaining meaningful classification.

Despite these contributions, there are challenges in effectively integrating classification across datasets that have different features, predicting minority classes and offering reasonable explanations behind automated decisions. The current technique tackles these challenges by using comparative evaluation on KC1 and PC1 benchmark datasets, improving prediction skills, and adding explainable artificial intelligence. Besides, a deployable interface enables easy application of defect estimation results, linking the predictive results with interpretable and usable software quality assessment.

3. Materials and Methods

The suggested approach intends to construct an effective software defect estimation framework with the use of KC1 and PC1 datasets from NASA PROMISE repository. Requires software quality indicators for detecting defect-prone modules, after the systematic classification process, data preparation, feature representation, model training and model performance evaluation. We include seven traditional ML algorithms as baseline algorithms: Bagging, RF, Multilayer Perceptron, Radial Basis Function, NB, MNB and SVM. Moreover, sophisticated gradient boosting approaches like XGBoost and CatBoost are employed, boosting the predictive power and capturing intricate relationships among software features. LIME and SHAP are added to offer local and global explanations to elucidate the forecast's transparency and interpretability. Each trained classification model is deployed in an easy-to-access interface using a FastAPI based deployment interface, so that the defect estimation can be accessed easily. The framework is designed to help in the reliable diagnosis of problems, the comparison of models, and software quality management.

DOI: https://doi.org/10.65477/ijmdas.2025.v1.i3.02



"Fig.1. System Architecture"

The overall software defect estimation system is illustrated in figure 1. The data sets KC1 and PC1 were retrieved from NASA PROMISE database and pre-processed prior to their input into the nine ML classification algorithms. They are examined by several indicators to forecast their performance. Local explanations and feature additions are provided for LIME and SHAP. The trained classification model is deployed with FastAPI to allow the user to feed software measurements and get back the output of the model, which predicts whether the software module is defective or otherwise.

A) Dataset Collection

In the method presented, the datasets are gathered from NASA PROMISE Repository. PROMISE Repository contains publicly available software defect information, for evaluation of software quality and software defect prediction techniques. Based on sample size, the properties of the software and the distribution of defective modules, we choose two benchmark datasets: KC1 and PC1.

KC1 Dataset: The KC1 dataset includes 2109 software modules with 22 parameters such as software complexity, code size and Halstead metrics. It has approximately 15.4% defective modules and 84.6% non-defective modules, which reflects the characteristics of software in C++. It has a moderately large number of samples so that defect classification is possible in comparison.

PC1 Dataset: This data set consists of 1109 software modules, and 22 parameters describing code complexity, programming structure and software quality measures. This software has 6.9% defective modules and 93.1% non-defective modules. It has a unique class distribution that can be used to test with heavy class imbalance.

B) Pre-Processing

The preprocessing phase consists of three steps: enhancing data consistency and arranging software quality attributes, analysing class distributions, and setting appropriate inputs to robust ML classification, that prepare the datasets KC1 and PC1 for robust ML classification.

Dataset Integration: The NASA PROMISE dataset is used for software defect estimation (KC1 and PC1 datasets). Both data sets are software quality measures which are representative of the complexity of the software, its structure and the features of its development. Each attribute and flaw label is given a comparison classification according to its specific attributes and flaw labels. The datasets are analysed separately, to keep the original properties and class distributions. This type of integration provides a uniform experimental foundation for the assessment of ML techniques for software fault prediction tasks.

Paper Available at: https://ijmdas.org
DOI: https://doi.org/10.65477/ijmdas.2025.v1.i3.02

Data Inspection: Structural properties, attribute distributions, data types and target class distributions of the selected data sets are checked. We focus in particular on the labels of the faulty and non-faulty modules, and on the software quality metrics. This review aims to find discrepancies, anomalies and differences between the data sets. It is vital that they can be understood so that an appropriate preprocessing technique can be selected, and a good foundation established for further categorisation and performance analysis of the systems.

Data Cleaning: Data has been cleaned to improve consistency and reliability of the software fault data selected. The records are checked for inconsistencies in attribute representation, incorrect observations and duplicate records. Any unwanted variation is straightened out to ensure quality of input data. This phase eliminates the repetition of input in the training of the model, and reduces possible distortion of the classification results. So the data sets after the cleaning offers the consistent view of the software attributes to be further used for predictive research.

Missing Value Handling: To maintain the completeness of software quality attributes used for categorisation, the management of missing values is examined. The characteristics of the data set are examined to determine if any of the observations are missing or incomplete, which may impact the performance of the predictive model. Selected features are processed appropriately to retain useful information and consistency amongst the selected features. This phase aims to prevent data incompleteness from interfering with the training/evaluation of the model. The quality of the data will be useful in making more valid comparisons among classification systems and greater consistency in predictive analysis.

Feature Preparation: The goal of feature preparation is to structure software quality attributes into a form suitable to be used in ML classification. The defect indicator is taken as the target property and the control-flow properties, cyclomatic complexity, Halstead characteristics and code size are kept as predictive factors. This separation is the connection between software attributes and defects being present. If features are selected properly in the software, classification algorithms can assess the characteristics of the software and identify trends in the defective and non-defective modules.

Feature Scaling: It helps to maintain numerical scale of software quality features equivalent. The data sets are selected to have metrics with different measurement units, including code-related counts, complexity assessments and Halstead values. Distances/gradients based learning approaches might be influenced by differences in numeric size. Normalization of the input attributes helps to minimize such variations and helps to train the model uniformly. This is especially true in algorithms like Support Vector Machine and Multilayer Perceptron where the size of features can alter the behaviour of predictions.

Class Distribution Analysis: The class distribution analysis explores the proportion of software modules with faults and non-faulty software modules in the KC1 and PC1 datasets. The percentage of bad modules in KC1 is approximately 15.4% and PC1 is around 6.9% of bad modules. These distributions show that there is class imbalance, especially in PC1. The minority class representation is critical to understand as classification models could tend toward the dominant class. This type of study can help in the sound evaluation and evaluation of the usefulness of models in the detection of defect-prone software modules.

Dataset Partitioning: Data is divided into various subgroups for training the model and test the model. The software metrics and the corresponding defect labels are arranged so that the input-output relationship is the same for each category of the defect labels. Stratified partitioning can maintain the proportions of faulty and non-defective modules in the subsets. This method helps to eliminate evaluation bias due to uneven class representation and gives a more representative evaluation of prediction capabilities. The generated partitions allow to compare systematically the nine specified categorisation techniques.

C) Algorithms

Bagging: Bagging is a method that uses multiple models, each of which is trained independently, to

Paper Available at: https://ijmdas.org
DOI: https://doi.org/10.65477/ijmdas.2025.v1.i3.02

get a more stable prediction and to diminish the variation. Produces predictions using numerous subsets of training data and merges their predictions. This results in more robust categorisation and more robust to the randomness of training data.

Random Forest: The RF generates lots of decision tree models on a random subset of training data and a random subset of features. Their individual predictions are combined to get the final classification. This provides greater stability in predictions, less overfitting, and captures complex relationships between input attributes.

Multilayer Perceptron (MLP): MLP is a type of feedforward network that is able to recognize complex relationships between layers of interconnected processing. It repeatedly optimises the weights and learns non-linear patterns of the input attributes and produces the classification outputs as output, to help with the problem diagnosis effectively.

Radial Basis Function (RBF): RBF networks use distance as an activation function to identify nonlinear relationships in the input data. They develop a non-linear mapping from the lower to the higher-dimensional representation of their deep characteristic layer that helps them succeed in the recognition of complicated patterns and flexibly solve prediction tasks.

Naive Bayes: NB is a probabilistic classification approach based on Bayes' Theorem and conditional independence assumption. It predicts the class distribution from the observed class values of characteristics and provides the best approximation of the class for the given data. It is fast to compute and provides a good initial classification.

Multinomial Naive Bayes: Predicts NB class probabilities from occurrences / frequencies of features. It calculates the probability for each category for each feature distribution. Hence it is probabilistic classifier. It is based on representation of the numerical input attributes.

Support Vector Machine (SVM): SVM computes an optimum decision boundary that optimally classifies the different classes by maximizing the distance between them. It is capable of learning the nonlinear classification patterns by means of kernel-based transformations and provide effective prediction capability in complex feature spaces.

Extreme Gradient Boosting (XGBoost): XGBoost is an advanced gradient boosting technique which adds decision tree models progressively to reduce the error of prediction. It employs regularisation and optimised learning algorithms to boost the classification performance, control the model complexity and learn the complex interrelationship between the input features.

Categorical Boosting (CatBoost): CatBoost is a gradient boosting technique that builds an ensemble of decision tree models in turn, rectifying the errors. It employs an ordered boosting algorithm and efficient feature processing techniques to improve the stability of the predictions, to reduce overfitting and to offer excellent classification accuracy in structured data.

4. Experimental Results

Accuracy: The "accuracy" of a test is its ability to correctly classify the patient as well as the healthy cases. To measure the accuracy of the test, the proportions of true positive and true negative should be computed in all cases tested. This can be mathematically expressed as:

$$"Accuracy = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{TN} + \mathrm{FN}} (1)"$$

Precision: Precision tells us how many of the instances or samples categorised as positives are classified accurately. Hence, the following is the formula for computing the precision:

$$"Precision = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} (2)"$$

Recall: Recall is a measurement in ML that shows how well a model is able to recognise all the relevant instances of a specific class. It is the ratio of the correctly predicted positive observations to the all observations in actual class. It gives insights on how thorough the model is in capturing the instances of a given class.

DOI: https://doi.org/10.65477/ijmdas.2025.v1.i3.02

$$"Recall = \frac{\text{TP}}{\text{TP} + \text{FN}} (3)"$$

F1-Score: F1 score is a metric of the accuracy of a ML model. It is an amalgamation of a model's accuracy and recovery rate. Accuracy statistic is the percentage of the time that the model predicted the result correctly on the entire data set.

$$"F1\ Score = 2 * \frac{Recall\ \text{X}\ Precision}{Recall + Precision} * 100 (4)"$$

AUC-ROC Curve: AUC-ROC Curve is a curve that provides a performance metric for classification at various thresholds. The ROC plot is a graph of True Positive Rate versus False Positive Rate. AUC is the overall performance of the model in the classification of the classes. The greater the AUC, the better the model.

$$"AUC = \sum_{i=1}^{n-1} (FPR_{i+1} - FPR_i) \cdot \frac{TPR_{i+1} + TPR_i}{2} (5)"$$

PR-AUC: PR-AUC is the area under the Precision-Recall curve and measures the performance of the classifier for various decision thresholds. It is especially beneficial for imbalanced datasets because it puts emphasis on the identification of minority-class instances.

$$"PR - AUC = \int_0^1 P(R) dR (6)"$$

MCC: In ML, MCC is a way to assess the performance of a binary classifier. It compares and analyzes the relationships between the predicted and actual binary results, considering all four elements of the confusion matrix.

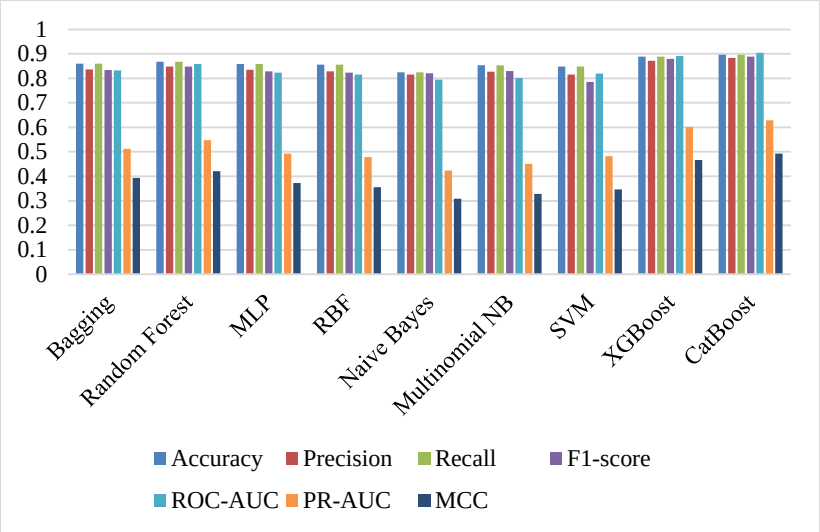
$$"MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN}} (7)"$$

"Table. 1. Performance Evaluation - KC1 dataset"

Algorithm	Accuracy	Precision	Recall	F1-score	ROC-AUC	PR-AUC	MCC
Bagging	0.860	0.836	0.860	0.834	0.832	0.512	0.394
Random Forest	0.867	0.848	0.867	0.848	0.858	0.548	0.421
MLP	0.859	0.835	0.859	0.828	0.823	0.493	0.372
RBF	0.856	0.828	0.856	0.823	0.816	0.478	0.356
Naive Bayes	0.824	0.816	0.824	0.820	0.795	0.423	0.309
Multinomial NB	0.853	0.827	0.853	0.830	0.801	0.451	0.328
SVM	0.848	0.816	0.848	0.786	0.819	0.482	0.347
XGBoost	0.889	0.871	0.889	0.879	0.891	0.601	0.467
CatBoost	0.897	0.883	0.897	0.889	0.904	0.628	0.493

Table I presents the results of the accuracy, precision, recall, F1-score, ROC-AUC, PR-AUC and MCC of nine ML methods applied to the KC1 dataset and indicates that CatBoost outperforms the other methods.

DOI: https://doi.org/10.65477/ijmdas.2025.v1.i3.02



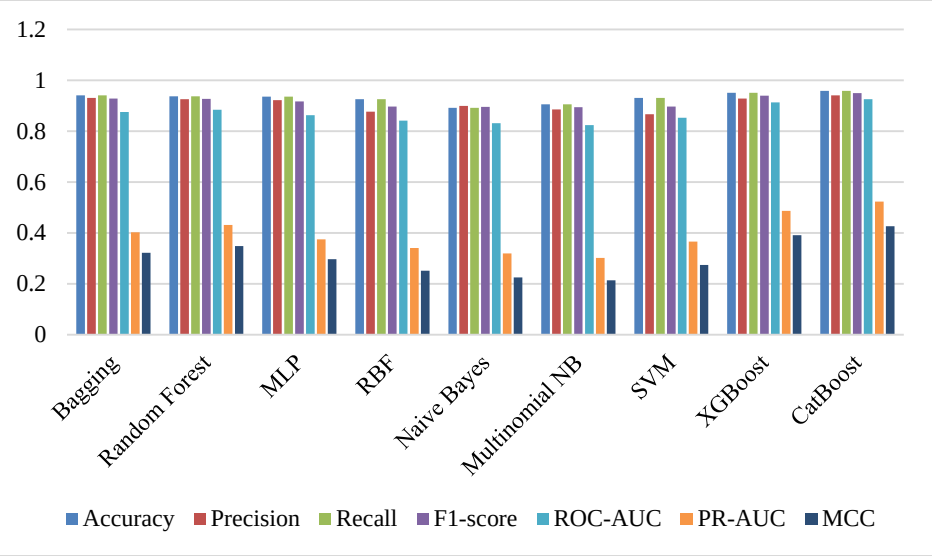
"Fig.2. performance comparison - KC1 dataset"

The performance comparison of nine ML algorithms using seven evaluation metrics are provided in 2. Catboost has the highest overall performance followed by XGBoost, which shows potential of both tools for software fault estimation.

"Table.2. Performance Evaluation - PC1 dataset"

Algorithm	Accuracy	Precision	Recall	F1-score	ROC-AUC	PR-AUC	MCC
Bagging	0.941	0.931	0.941	0.928	0.875	0.402	0.322
Random Forest	0.937	0.925	0.937	0.927	0.884	0.431	0.348
MLP	0.936	0.921	0.936	0.917	0.862	0.375	0.296
RBF	0.926	0.876	0.926	0.896	0.841	0.341	0.251
Naive Bayes	0.892	0.899	0.892	0.895	0.831	0.319	0.225
Multinomial NB	0.905	0.885	0.905	0.894	0.824	0.302	0.213
SVM	0.930	0.866	0.930	0.897	0.853	0.366	0.274
XGBoost	0.951	0.928	0.951	0.939	0.913	0.486	0.391
CatBoost	0.958	0.941	0.958	0.949	0.926	0.523	0.426

Table 2 shows that nine ML methods were evaluated by seven metrics in the PC1 dataset. The system with the highest accuracy is CatBoost with 95.8% precision, which demonstrates the most accurate categorisation performance prediction.



"Fig.3. performance comparison - PC1 dataset"

The performance of the nine ML methods is compared with respect to the seven assessment criteria in image dataset PC1. The accuracy and F1-score are the highest in Catboost. Xgboost has competitive classification performance.

5. Conclusion

In conclusion, it is hoped that the proposed software defect estimation framework will help in the effective maintenance decision making and correctly identify the modules which are prone to software defect and thus improve the software quality assurance. The approach is based on the methodology presented in the KC1 and PC1 datasets provided in NASA PROMISE repository and uses the software quality criteria to assess the fault classification performance. We use seven classical ML techniques: Bagging, RF, MLP, Radial Basis Function, NB, MNB and SVM to compare with the advanced ones: XGBoost and CatBoost. The accuracy of the software module identification is obtained from the estimates of the provisional evaluation: the highest accuracy is achieved by CatBoost with accuracy of 89.7% on KC1 and 95.8% on PC1. Furthermore, the integration of LIME and SHAP enhances the interpretability of the model, providing insights into individual predictions and key features of the software. FastAPI deployment interface is utilised to access the model to anticipate software defects in an application context. We propose a single framework for software defect estimates by integrating comparative categorisation, sophisticated ensemble learning, explainable artificial intelligence and API-based deployment. In summary, this approach helps in a more systematic identification of defects, provides interpretable prediction results, and helps in the effective management of the quality of software, which leads to informed software maintenance decisions and enhancement of software reliability.

To improve the proposed framework further, additional collections of software defects from other programming language(s) and development environment(s) could be added to expand the generalisation. Some of the other advanced ensemble and hybrid learning methods could be explored to further enhance the classification accuracy and detection of minority classes. It's possible to connect automated hyperparameter optimization, making your model more efficient and more consistent in its predictions. Moreover, LIME and SHAP can be enhanced to give new insights into software quality indicators. FastAPI can be deployed to the cloud to enable scalability for integration with software development and continuous testing pipelines.

References

1. V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Transactions on Software Engineering, vol. 22, no. 10, pp. 751–761, Oct. 1996, doi: 10.1109/32.544352.
2. E. Ceylan, F. O. Kutlubay, and A. B. Bener, "Software defect identification using machine learning techniques," in Proc. 32nd EUROMICRO Conf. Software Engineering and Advanced Applications (EUROMICRO-SEAA), 2006, pp. 240–247, doi: 10.1109/EUROMICRO.2006.56.
3. K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," Journal of Systems and Software, vol. 81, no. 5, pp. 649–660, May 2008, doi: 10.1016/j.jss.2007.07.040.
4. L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in Proc. 15th International Symposium on Software Reliability Engineering (ISSRE), 2004, pp. 417–428, doi: 10.1109/ISSRE.2004.35.
5. T. M. Khoshgoftaar, E. B. Allen, and J. Deng, "Using regression trees to classify fault-prone software modules," IEEE Transactions on Reliability, vol. 51, no. 4, pp. 455–462, Dec. 2002, doi: 10.1109/TR.2002.804488.
6. T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and S. J. Aud, "Application of neural networks to software quality modeling of a very large telecommunications system," IEEE Transactions on Neural Networks, vol. 8, no. 4, pp. 902–909, Jul. 1997, doi: 10.1109/72.595888.
7. S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in Proc. 33rd International Conference on Software Engineering (ICSE), 2011, pp. 481–490, doi: 10.1145/1985793.1985859.
8. Y. Ma, L. Guo, and B. Cukic, "A statistical framework for the prediction of fault-proneness," in Advances in Machine Learning Applications in Software Engineering, 2007, pp. 237–263, doi: 10.4018/978-1-59140-941-1.ch010.
9. R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," Applied Soft Computing, vol. 27, pp. 504–518, Feb. 2015, doi: 10.1016/j.asoc.2014.11.023.
10. J. Ren, K. Qin, Y. Ma, and G. Luo, "On software defect prediction using machine learning," Journal of Applied Mathematics, vol. 2014, Art. no. 785435, 2014, doi: 10.1155/2014/785435.
11. S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," IEEE Transactions on Reliability, vol. 62, no. 2, pp. 434–443, Jun. 2013, doi: 10.1109/TR.2013.2259203.
12. S. Hussain, J. W. Keung, A. A. Khan, and K. E. Bennin, "Performance evaluation of ensemble methods for software fault prediction: An experiment," in Proc. 24th Australasian Software Engineering Conference (ASWEC), 2015, pp. 91–95, doi: 10.1145/2811681.2811699.
13. M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in Advances in Neural Information Processing Systems 28 (NeurIPS), 2015, pp. 2962–2970.
14. K. Tanaka and A. Monden, "Empirical evaluation of bug prediction using auto-sklearn," in Proc. JSSST Workshop on Foundations of Software Engineering (FOSE), 2018, pp. 31–40.
15. K. Tanaka, A. Monden, and Y. Zeynep, "Effectiveness of auto-sklearn in software bug prediction," Computer Software, vol. 36, no. 4, pp. 46–52, 2019, doi: 10.11309/jssst.36.4_46.
16. Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in Proc. 26th IEEE International Conference on Software Maintenance (ICSM), 2010, pp. 1–10, doi: 10.1109/ICSM.2010.5609530.
17. S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," IEEE Transactions on Software Engineering, vol. 34, no. 4, pp. 485–496, Jul.–Aug. 2008, doi: 10.1109/TSE.2008.35.
18. K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, "MAHAKIL: Diversity

based oversampling approach to alleviate the class imbalance issue in software defect prediction," IEEE Transactions on Software Engineering, vol. 44, no. 6, pp. 534–550, Jun. 2018, doi: 10.1109/TSE.2017.2731766.

19. W. Fu, V. Nair, and T. Menzies, "Why is differential evolution better than grid search for tuning defect predictors?" arXiv preprint arXiv:1609.02613, 2016. [Online]. Available: https://arxiv.org/abs/1609.02613.
20. A. Monden, J. Keung, S. Morisaki, Y. Kamei, and K. Matsumoto, "A heuristic rule reduction approach to software fault-proneness prediction," in Proc. 19th Asia-Pacific Software Engineering Conference (APSEC), 2012, pp. 838–847, doi: 10.1109/APSEC.2012.103.