Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Transparency in Software Defect Prediction: A Dual Approach using Explainability and Tradeoff Analysis

Nitin Sai Bommi
University of Southern California
Umamaheswara Sharma Bhutamapuram
busharma@nitc.ac.in
National Institute of Technology Calicut
Atul Negi
University of Hyderabad

Research Article

Keywords: Software Defect Severity Prediction, Software Quality, Performance Evaluation Measures, Machine Learning

Posted Date: August 4th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10576352/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Transparency in Software Defect Prediction: A Dual Approach using Explainability and Tradeoff Analysis

Nitin Sai Bommi[1], Umamaheswara Sharma B[2*], Atul Negi[3]

[1]Master of Science, Viterbi School of Engineering, University of Southern California, Los Angeles, California, California, U.S.A.
[2]Assistant Professor, ASER Lab, National Institute of Technology Calicut, Kattangal, Kozhikode, 673601, Kerala, India.
[3]Professor, School of Computer and Information Sciences, University of Hyderabad, Hyderabad, Hyderabad, 500046, Telangana, India.

*Corresponding author(s). E-mail(s): busharma@nitc.ac.in;
Contributing authors: nitinsai.bommi@gmail.com; atulnegi@uohyd.ac.in;

Abstract

Software Defect Prediction (SDP) is a critical task in software engineering, aiming to identify potential defects in software systems before they become costly. However, current models mainly focus on developing new methods and feature engineering techniques to improve accuracy, which is not the best metric for imbalanced datasets. In this work, we formulate the problem as maximizing the difference between the probability of detection and the probability of false alarm, two important metrics in the model training phase. To solve the problem, we propose two possible solutions: one is a new objective for SDP that aims to minimise the difference between the two metrics by adjusting the threshold value and the other is using counterfactual explanations to generate new data points, and fine-tuning the model on the new data to improve the performance and collaterally, the difference between the metrics. Our experiments demonstrate that the proposed methods outperform the traditional fixed threshold approach. These approaches can contribute to developing more accurate and reliable classifiers for real-world software defect prediction applications, especially in imbalanced datasets. Our proposed methods provide a more balanced approach, which finds an optimal trade-off between minimizing false positives and false negatives.

Keywords: Software Defect Severity Prediction, Software Quality, Performance Evaluation Measures, Machine Learning

1 Introduction

Software defect prediction (SDP) is a machine learning (ML)-based classification task targeted to help software testers by automatically providing the defect-proneness of newly developed software modules, in a quick time Herbold et al. (2017); Tantithamthavorn et al. (2017, 2018); B and Sadam (2023). Therefore, the software quality assurance (SQA) team may focus their limited resources only on the predicted defective modules, rather than spending time on testing each developed software module. Typically, the SDP models use the metrics information of each software program as independent features in the training and testing processes. In general, the mined historical data from the software projects is used to train the prediction models. Later, using the trained prediction model, the defect-proneness of a newly developed software program is determined Zimmermann et al. (2009); Ma et al. (2012); Mizuno and Hirata (2014); Xia et al. (2016); Herbold et al. (2017); Wu et al. (2018); Xu et al. (2019); Jin (2021); B and Sadam (2023).

While the existing literature has made considerable strides in addressing fundamental challenges related to prediction model construction, including topics such as selecting appropriate model validation techniques to reduce bias during the development of the model Tantithamthavorn et al. (2017), the impact of automated parameter optimisation Tantithamthavorn (2016), and overarching methodologies for practitioners to consider Lessmann et al. (2008); Tantithamthavorn and Hassan (2018), among others, it remains clear that prediction models often fall short in delivering optimal predictive performance Herbold et al. (2017). Additionally, as emphasized in a study by Sharma et al. B and Sadam (2023), the primary goal of predictive models is to provide insights into project objectives. Specifically, these models aim to maximise budget savings while minimising unfortunate project failures. Although these objectives are paramount, achieving them requires a delicate balance between two critical metrics: the probability of detection (PD) and the probability of false alarm (PFA).

Observing the right trade-off between PD and PFA plays an important role and is an influential factor in observing the final prediction performances Shalev-Shwartz and Ben-David (2014); Bishop and Nasrabadi (2006); Guyon and Elisseeff (2003). In general, increasing the probability of detection often leads to increased false alarms, while reducing false alarms may result in missing some actual defects Bishop and Nasrabadi (2006). This trade-off involves finding the right balance between accurately identifying (or maximising) defects (high probability of detection) and minimising false positives (low probability of false alarm), ensuring the increased budget savings and the reduced future failure incidents. Hence, the performance of the prediction model is largely dependent on finding the right threshold at which the gap between the probability of detection and the probability of false alarm is maximum James et al. (2013).

Though most literature focuses on improving a single measure (for example: accuracy, precision, recall, or f-measure), several studies claim that the use of traditional measures may not provide sufficient information about the predictions B and Sadam (2023); Morasca and Lavazza (2020). We need to consider two possible scenarios where these measures may not accurately reflect the actual performance of the developed prediction models. As mentioned before, if the number of false negatives (FNs) rises,

the test team might overlook the defect, which, in the worst-case scenario during operation, could result in a catastrophe Lyu et al. (1996). On the other hand, if the number of false positives (FPs) increases, the test-team may have to invest significant time and resources into debugging clean code B and Sadam (2023). Therefore, it is crucial to balance these two scenarios when developing effective prediction models. However, if we try to reduce FPs, the number of FNs will increase and vice versa. To tackle this situation, we propose a new objective that aims at minimising both FPs and FNs by adjusting the threshold iteratively. To find an optimal trade-off, we formulate the problem as maximising the difference between the probability of detection and the probability of a false alarm.

Furthermore, analysing the behaviour of the developed prediction model on misclassified programs may reduce the burden on tester Molnar (2020). For critical ML applications like SDP, analysing the behaviour of the developed model helps to minimise the misclassifications and thus to improve the model performance Molnar (2020); Tantithamthavorn et al. (2023). To analyse the behaviour of the model, in this work, we also targeted to generate synthetic data using counterfactual explanations from the trained prediction model, majorly focusing on the true positive (TP) and true negative (TN) instances. Specifically, we generate counterfactual explanations for both the true positives and true negatives, as the model correctly classifies these test instances. Since the model understood the pattern of the data, we generate the counterfactuals with the appropriate opposite class. That is, we generate counterfactual explanations for the appropriate class as clean for some of the true positive instances and defective for some of the true negative instances. This is an iterative process, targeted to improve the model's performance and get insights into the predictions.

We conducted experiments using the probabilistic ML models such as Logistic Regression, Naïve Bayes, and Neural Networks using our iterative approach for finding the optimal threshold. The results of individual techniques across Logistic Regression, Naïve Bayes, and Neural Network models indicate that using an optimal threshold approach consistently improved performance. For Logistic Regression, the average optimal threshold was 0.35, and reduced False Omission Rate (FOR) measure, with a notable improvement at a threshold of 0.4. The Naïve Bayes model performed best at an average threshold of 0.3, showing significant differences between pd and pda, and lower FOR values. The Neural Network model had an average optimal threshold of 0.38, achieving the highest metric differences and benefiting from counterfactual data, though FOR was lowest with the counterfactual approach. Across all models, selecting optimal thresholds rather than the default 0.5 led to better performance, highlighting the value of threshold tuning in model evaluation.

To the best of our knowledge, in this work, we make the following unique contributions.

1. As pre-defined threshold values for the target example may lead to biased estimates, we proposed an iterative approach that selects an optimal threshold value, targeting to improve the performance of SDP.
2. We investigated the use of Counterfactual explanations targetting to improve the performance of the SDP. These explanations help developers and stakeholders

comprehend what changes would alter the prediction, providing insights into how specific features or attributes contribute to the prediction of software defects.
3. A large-scale empirical evaluation and comparative analysis is carried out using the proposed optimal threshold, counterfactual, and other default threshold approaches on 36 various releases of 10 software projects.

***Paper Organisation*:** The rest of the paper is organised as follows. Section 2 discusses the recent approaches to our problem statement. In Section 3, we modeled the problem of finding the right balance between the probability of detection and the probability of a false alarm and discusses a detailed methodology of the proposed optimal approach threshold and an approach for finetuning the model using counterfactual explanations. A detailed description of the datasets, probabilistic ML models, performance measures, etc. is presented in Section 4. In Section 5, we present the results of our conducted study, the comparative analysis, and a detailed discussion of the potential threats that may validate the performance of the proposed solution approaches. The conclusion and possible potential future directions from our approach are presented in Section 6.

2 Related Work

In this section, we provide the literature that focuses majorly on the techniques that asses the prediction models, particularly the works focuses on reducing the false alarms and the probability of detection while maintaining the highest performances.

It is evident that many defect prediction tasks in the literature do focus on solving various problems in relation to the developement of the prediction models and very few considered reducing the false alarms Tosun and Bener (2009). Yet we can observe that the literature least focuses on finding the right threshold in order to estimate a best-fitted hypothesis that minimises the misclassification on both the training and the test data. As specified in Tosun and Bener (2009) by Tosun and Bener, the defect prediction models are prone to produce high misclassifications as the input data may contain an imbalance distribution of the defective classes. Also, since the cost of misclassifications of defective modules is critical, they suggest that the choice of using the right threshold results in the best separation of the defective and defect-free modules. This essentially reveals the negative effect of using default threshold settings in the probabilistic classifiers.

On the other hand, it is evident that the advent of explainability improves the performance of the defect prediction models Tantithamthavorn and Jiarpakdee (2021); Bommi and Negi (2023); Tantithamthavorn et al. (2023). Tantithamthavorn et al. in Tantithamthavorn et al. (2023) provide a detailed analysis of the role of eXplainable Artificial Intelligence (XAI) in defect prediction tasks. As described in Tantithamthavorn et al. (2023), these XAI techniques are contributing to more efficient software development processes, leading to improved outcomes Tantithamthavorn and Jiarpakdee (2021). However, the lack of transparency makes it difficult for practitioners to understand why specific commits are considered risky and how they can be improved Khanan et al. (2020). In addition, the adoption of JIT defect prediction

has been limited because it has not been consistently integrated into continuous integration/continuous deployment (CI/CD) pipelines and modern software development platforms, such as GitHub Khanan et al. (2020). To address these issues, Chaiyakarn Khanan et al. in Khanan et al. (2020) presents an explainable Just-In-Time (JIT) defect prediction framework called the JITBot, not only to predict the riskiness of individual commits but also provide explanations for why a particular commit is considered risky. It goes a step further by suggesting risk mitigation plans to help developers address potential issues. Importantly, this framework is integrated into the GitHub CI/CD pipeline as a GitHub application, enabling continuous monitoring and analysis of commits in multiple GitHub repositories. The paper also discusses different scenarios in which this framework can be used by software development practitioners, emphasizing its potential to enhance software quality and developer productivity.

In Jiarpakdee et al. (2021), Jiarpakdee et al. aim to understand how practitioners perceive these goals and the model-agnostic techniques used to create visual explanations of defect prediction models. A significant majority of respondents (82%-84%) found the three common goals of defect prediction models to be useful. The Local Interpretable Model-Agnostic Explanations (LIME), an XAI technique, was the most preferred method for understanding the most important characteristics that contribute to the prediction of defects in the software modules. Analysis of Variance (ANOVA)/VarImp was the second most preferred technique for understanding the characteristics associated with software defects in the past. These findings emphacise the importance of improving the understanding of defect prediction models and their predictions among practitioners. Model-agnostic techniques are suggested as valuable tools to help practitioners gain insights into defect prediction models and enhance their utility in software development processes.

On a similar objective, Jiarpakdee et al. in Jiarpakdee et al. (2020) conducted an empirical evaluation of three model-agnostic techniques: two widely used methods, LIME and iBreakDown, and an enhanced version of LIME with Hyper Parameter optimisation (LIME-HPO). The study involves 32 curated defect datasets from nine open-source software systems. The key findings of the study include the need for model-agnostic techniques to explain individual predictions of defect models, as current models lack this capability, instance explanations generated by model-agnostic techniques are mostly consistent (although not exactly the same) with the global explanations of defect models. These explanations are reliable when regenerated. They suggested that the model-agnostic techniques are efficient, taking less than a minute to generate instance explanations. More than half of the practitioners surveyed find contrastive explanations, which highlight differences between predictions, necessary and useful for understanding defect predictions Tantithamthavorn and Jiarpakdee (2021).

In Aleithan (2021), Reem Aleithan addressed the challenge of explaining the results of software bug prediction models, particularly in the context of just-in-time (change-level) bug prediction. The authors reference the work by Jirayus et al. Jiarpakdee et al. (2020), who proposed the use of two model-agnostic techniques, LIME, and iBreakDown to explain the results of bug prediction models at the file level. While promising results were achieved in file-level bug prediction, Reem Aleithan found that the effectiveness of these techniques in explaining just-in-time bug prediction results

remained unknown. To address this gap, Reem Aleithan conducted the first empirical study focused on exploring the explainability of model-agnostic techniques, specifically LIME and iBreakDown, in the context of just-in-time bug prediction models. The study follows a three-step approach: 1. replicating widely used just-in-time bug prediction models, 2. applying LIME and iBreakDown to the prediction results of these models, and 3. manually evaluating the explanations provided by these techniques for instances where bugs were predicted (positive predictions), comparing them against the actual root causes of the bugs. The results of the experiment indicate that LIME and iBreakDown are not effective in explaining defect predictions for just-in-time bug prediction models, in contrast to their performance at the file level. This finding underscores the need for new approaches and techniques to improve the explainability of just-in-time bug prediction models.

Though all the listed models try to improve the performance of an SDP model either by the traditional approaches or by the explainability, to observe the right improvement in the performance, however, they missed to focus on bringing balance between the probability of detection and probability of false alarm. Hence, it is essential to focus more on finding the trade-off between the detection and the false alarms. To bridge this gap, as mentioned above, we proposed an iterative and a counterfactual explanation-based approachs for software defect prediction. A detailed formulation of our approachs are presented in the following section.

3 The Problem and the Methodology

In this section, we first present the problem of finding the optimal threshold, targeting to improve the prediction accuracy. Also, we discuss a detailed methodology to solve the framed problem using both an iterative approach and the counterfactual explanations.

3.1 Problem Formulation

In ML literature, accuracy is a prevalent metric for assessing a model's performance. Nevertheless, when evaluated on imbalanced data, it fails to capture the essence of the model's capabilities. To illustrate this point, consider a scenario with 9 clean modules for every 1 defective module. If a dummy classifier is trained to invariably predict clean, we would still obtain a classifier with an accuracy of 90%. However, upon examining alternative metrics such as recall and F1-score, it becomes evident that the model's performance is worse. Hence, an ideal model attains optimal values across various metrics.

As specified by Sayyad Shirabad et al. in Sayyad Shirabad and Menzies (2005), it is observed that an ideal classifier will have a high probability of detection and a low probability of false alarm. Instead of focusing on optimising one metric, we aim to design the problem such that the algorithm performs better on both metrics (probability of detection and probability of false alarm). A classifier with such metric values - high probability of detection and low probability of false alarm - will also have a high difference between them. Instead of optimising one metric, our objective is to increase the distance between *Pr[Detection]* and *Pr[FalseAlarm]*, which in turn makes sure that

the individual metrics are optimal. Here, $Pr[*]$ is a probability function. Throughout the discussion, we use pd and pfa, interchangebly with the terms *Pr[Detection]* and *Pr[FalseAlarm]*, respectively. Below, we represent the objective function of any classifier, that is targeted to maximise the difference between the probability of detection and the probability of false alarm.

$$max\ \{Pr[Detection]\ -\ Pr[FalseAlarm]\} \tag{1}$$

where,

$$Pr[Detection] = \frac{TP}{TP + FN} \tag{2}$$

$$Pr[FalseAlarm] = \frac{FP}{TN + FP} \tag{3}$$

Another approach to tackle this imbalanced data would be to increase the minority class instances. However, we cannot rely on standard synthetic data generation techniques like synthetic minority oversampling technique (SMOTE) and its other variants for these applications as they can be risky Ramezankhani et al. (2016); Blagus and Lusa (2013); Agrawal and Menzies (2018). We need to leverage the existing data points to generate new data points that help us balance the dataset. A different strategy is to make use of the counterfactual explanations to generate the data points. A detailed methodology of the counterfactual explanations is given in the following section.

3.2 Methodology

As formulated a problem statement in Section 3.1, we framed an optimisation function that solves the problem in an efficient and robust way. To address Equation 1, we propose two techniques. These 2 techniques solve the objective in two different ways. Algorithms 1 and 2 describe the two proposed approaches. A detailed methodology of the two methods is given below.

3.2.1 Exploring the Optimal Threshold

In this technique, we change the decision threshold to solve the objective. To change the decision threshold, we consider probabilistic ML models that output a probability score of a instance.

For a model, assume θ is the decision threshold and p is the probability with which the output is predicted, then:

$$Prediction = \begin{cases} 1\ (Defective) & if\ p > \theta \\ 0\ (Clean) & Otherwise \end{cases} \tag{4}$$

Here, assume that, class 1 as the defective and 0 as the clean module. By default, 0.5 is used as the decision threshold (θ) in many ML models to classify the instances. Instead of having a static θ value, we can make it dynamic to adapt to the testing environment by solving the following objective function:

$$\max_{\theta}\ \{Pr[Detection] - Pr[FalseAlarm]\} \tag{5}$$

Such that, $0.1 \leq \theta \leq 0.9$.

It is obvious that, when the value of θ is 0, it classifies everything as belonging to defective, thereby increasing the number of false positives. Similarly, 1, it classifies everything as clean, leading to many false negatives. Moreover, at these two extremes, the model becomes a dummy classifier that predicts the same output irrespective of the input. We propose the θ that satisfies Equation 5 as the optimal threshold and consider the model at that threshold as an ideal model since it has high probability of detection and low probability of false alarm.

In the testing phase of the model, while evaluating the software modules, the model provides probabilities indicating the likelihood that a particular instance is defective. These probabilities are numerical values between 0 and 1, with 1 representing a high likelihood of being defective. To determine the final classification of each instance, we employ the threshold-based approach as proposed above. If the probability of the output by the model is greater than or equal to θ, the instance is classified as defective; otherwise, it is classified as clean.

By varying the threshold value, we can obtain different output classes for the instances. For instance, if we decrease the threshold below 0.5 (0.3), more instances will be classified as defective, thereby reducing the number of false negatives. On the other hand, increasing the threshold above 0.5 (0.7) may result in fewer instances being classified as defective, reducing the number of false positives. By systematically adjusting the threshold within a specified range, we can generate a set of output classes that reflect different trade-offs between FPs and FNs. This flexibility allows us to tailor the model's behavior to specific requirements or constraints in our application. We extract the confusion matrix from the different output classes, which gives information about TP, TN, FP and FN. From these metrics, we calculate equations 2 and 3 to estimate the best suitable dynamic θ value.

We utilised probabilistic output models to validate our proposed approach, namely Logistic Regression, Naïve Bayes, and Neural Networks. A cross-project approach was employed to train the models, where one dataset was held out for testing while the remaining datasets were used for training. Following the training phase, the models were evaluated by generating probability scores for the test instances.

The θ parameter, representing the threshold, is varied from 0.1 to 0.9 with an increment of 0.1 to compute the difference between the proposed metrics in equations 2 and 3. The optimal threshold is determined by selecting the θ value that yielded the maximum difference. Other relevant metrics were reported at the selected threshold. The complete process of obtaining the optimal threshold is illustrated in Algorithm 1. By utilizing this process, we aimed to minimise false positives and negatives rather than solely focusing on accuracy, which is not the best metric for imbalanced datasets.

Algorithm 1 outlines the steps to find the optimal threshold for the proposed objective. The algorithm takes a dataset containing instances from the training distribution as input and returns the optimal threshold. The algorithm initialises the threshold parameter to zero and sets the initial difference to zero. It also creates an empty list to store the predicted class labels for each instance in the dataset. As mentioned above, the algorithm iterates over a range of threshold values from 0.1 to 0.9 with a step of

Algorithm 1: An Algorithm to Get the Optimal Threshold

```
Initialize θ_optim and max_difference to 0 ;
Initialize predicted_class as [] ;
for θ in range [0.1, 0.9] with a step of 0.1 do
    for Every instance in the dataset do
        Find the probability (p) of instance belonging to defective class ;
        if p > θ then
            Insert 1 into predicted_class array ;
        else
            Insert 0 into predicted_class array ;
        end
    end
    Calculate TP, FP, TN, FN ;
    Calculate pd and pfa ;
    if (pd − pfa) > max_difference then
        Set θ_optim to θ ;
        Update max_difference = (pd − pfa)
    end
end
```

0.1. For each threshold value, it predicts the class label for each instance in the dataset using the trained model and the current threshold. If the probability of the instance being in a positive class is greater than the threshold, it is classified as a positive instance; otherwise, it is classified as a negative instance. The predicted class labels are stored in the list.

After predicting the class labels for all instances in the dataset, the algorithm computes the confusion matrix for the predicted class labels and the real class labels. Using the confusion matrix, it calculates the probability of detection (pd) and the probability of false alarm (pfa) for the current threshold value. The algorithm then checks whether the difference between pd and pfa at the current threshold is greater than the current maximum difference. If so, it updates the maximum difference and sets the optimal threshold to the current threshold value. Finally, the algorithm returns the optimal threshold value. This threshold is used to evaluate the model's performance on the test dataset, and various metrics are computed at this threshold value. The algorithm serves as a useful tool for finding the optimal threshold value that maximises the difference between the probability of detection and the probability of false alarm, which is crucial for developing effective software defect prediction models.

3.2.2 Finetune Model With Counterfactual Explanations

In addition to our optimal threshold approach for improving the model's performance, in this work, we also make use of counterfactual explanations to generate synthetic data. We propose this technique based on the assumption that the dataset is imbalanced with few defective instances. In general, if the dataset is imbalanced, eventually

the model will learn biasedly towards the majority class (in this case, the clean class). As specified in previous sections, this is the major reason to have more probability of detection and false alarms. Hence, we need to generate more synthetic data to balance both classes. To do this, after training the model, we generate counterfactual explanations for TPs and TNs, as the instances are correctly classified by the model. We generate counterfactual explanations with the desired opposite class, i.e., we generate counterfactuals with the desired class as clean for TP instances and defective for TN instances. A detailed approach to finetuning the model using counterfactual explanations is given in Algorithm 2.

Algorithm 2: Finetune Model using Counterfactual Explanations

```
Train the model on the training data ;
Calculate TP, FP, TN, FN on the test data ;
for Every TP and TN do
    Generate n-counterfactuals that flip the prediction ;
end
Create a new dataset with counterfactuals and flipped outcomes ;
Finetune the trained model on this new data ;
for Every TP and FP do
    Generate counterfactual examples to aid developers in solving the defect ;
end
```

Algorithm 2 aims to enhance the performance of a trained model by leveraging counterfactual explanations. Counterfactual explanations refer to instances or examples that, when modified slightly, result in a different prediction outcome. The algorithm follows a series of steps to achieve this objective.

First, the algorithm begins by training the model on the available training data. Once the training is complete, it proceeds to evaluate the model's performance on the test data by calculating the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Next, the algorithm generates a set of counterfactuals for every TP and TN instance. These counterfactuals are instances that have their features modified in such a way that the model's prediction flips. This process helps to explore different scenarios and understand the factors that contribute to the model's predictions. Subsequently, a new dataset is created by combining the original data with the generated counterfactual instances, and the outcomes of these instances are flipped. This augmented dataset is then used to finetune the previously trained model. Finetuning involves updating the model's parameters using the new data to improve performance. After finetuning, the algorithm focuses on TP and FP instances. For each TP and FP, it generates a counterfactual example with the intention of assisting developers in solving the identified defect. These counterfactual examples provide insights into how the model's predictions can be changed by modifying certain features, aiding in the understanding and debugging potential issues.

In a nutshell, the algorithm starts by training a model and evaluating its performance. It then generates counterfactuals for TP and TN instances, creating a new

dataset and finetuning the model. Finally, counterfactual examples are generated for TP and FP instances to help developers address defects. This iterative process enhances the model's performance and provides valuable explanations and insights into its predictions.

4 Empirical Setup

In this section, we provide a detailed description of the utilized defect datasets for the empirical evaluation, the specific SDP task called the cross-project defect prediction (CPDP), the utilized base probabilistic classifiers for the task of CPDP, the performance measures, and the statistical significance tests, in the sections, 4.1, 4.2, 4.3, 4.4, and in 4.5, respectively.

4.1 Benchmark Datasets

To validate our approach, we used the defects data of various software projects from the publicly available PROMISE repository. Table 1 gives a detailed description of the data set. For the empirical study, we utilized 10 projects along with their released versions. In total, we have used a total of 36 versions from the 10 homogeneous software projects. For each software project, we present the number of modules, the lines of code (LoC), the number of defects, and the percentage of defective modules.

4.2 The Utilised SDP Task

The research on SDP further sub-divided majorly into within-project defect prediction (WPDP) Bhutamapuram and Sadam (2022), cross-project defect prediction (CPDP) Herbold (2013); Lessmann et al. (2008), and heterogeneous defect prediction (HDP) Nam et al. (2017). As the literature majorly focussed on the CPDP task, we use the same to conduct the empirical evaluation. Nevertheless, we can validate our objective using other tasks such as WPDP, HDP, JIT-DP, etc.

For the empirical analysis, we followed the general CPDP architecture wherein we used the defects data of all the source projects (including their past releases), except the target project, to train a prediction model. Later, the defect-proneness of the modules in the latest version of the target project will be found using the trained CPDP model.

4.3 Probabilistic Classifiers

The methodology presented in section 3.2 involves determining an appropriate threshold for grouping predictions into defective and clean categories. This necessitates that the output of the model be a probability score, represented as a decimal within the range of 0 to 1. Our approach utilized models that generate such probability scores to facilitate this.

In this study, the Logistic Regression model is utilized due to its ability to utilize a sigmoid function to confine the output within the range of (0, 1), which is appropriate for the experimental design. Similarly, the Neural Network model applies sigmoid or tanh activation functions for binary classification purposes. Specifically, for this

Table 1: A description of the utilized software projects

Project	Modules	LoC	Defects	%Defects	Project	Modules	LoC	Defects	%Defects
PROMISE Projects									
Ant-1.3	125	37,699	20	16.00	Lucene-2.2	247	63,571	144	58.3
Ant-1.4	178	54,195	40	22.47	Lucene-2.4	340	102,859	203	59.71
Ant-1.5	293	87,047	32	10.92	Poi-1.5	237	55,428	141	59.49
Ant-1.6	351	113,246	92	26.21	Poi-2.0	314	93,171	37	11.78
Ant-1.7	745	208,653	166	22.28	Poi-2.5	385	119,731	248	64.42
Camel-1.0	339	33,721	13	03.83	Poi-3.0	442	129,327	281	63.57
Camel-1.2	608	66,302	216	35.53	Synapse-1.0	157	28,806	16	10.19
Camel-1.4	872	98,080	145	16.63	Synapse-1.1	222	42,302	60	27.03
Camel-1.6	965	113,055	188	19.48	Synapse-1.2	256	53,500	86	33.59
Jedit-3.2	272	128,883	90	33.09	Velocity-1.4	196	51,713	147	75.00
Jedit-4.0	306	144,803	75	24.51	Velocity-1.6	229	57,012	78	34.06
Jedit-4.1	312	153,087	79	25.32	Xalan-2.4	723	225,088	110	15.21
Jedit-4.2	367	170,683	48	13.08	Xalan-2.5	803	304,864	387	48.19
Jedit-4.3	492	202,363	11	02.24	Xalan-2.6	885	411,737	411	46.44
Log4j-1.0	135	21,549	34	25.19	Xalan-2.7	909	428,555	898	98.79
Log4j-1.1	109	19,938	37	33.95	Xerces-1.2	440	159,254	71	16.14
Log4j-1.2	205	38,191	189	92.20	Xerces-1.3	453	167,095	69	15.23
Lucene-2.0	195	50,596	91	46.67	Xerces-1.4	588	141,180	437	74.32

research, a Neural Network model with a sigmoid activation function is used in the final layer. On the other hand, the Naïve Bayes model offers a probabilistic output by default, without the need for any additional processing, unlike the logistic regression and neural network models, which need a sigmoid function to convert the output to a probabilistic value. This unique feature of the Naïve Bayes model makes it an ideal choice for this experiment.

4.4 Performance Measures

To evaluate the performance of our approach, we used the following metrics proposed by Sharma et al. in B and Sadam (2023):

4.4.1 False Omission Rate (FOR)

False Omission Rate is a metric that quantifies the proportion of actual defects that are incorrectly classified as clean by a prediction model. Mathematically, FOR is defined as the ratio of the number of false negatives to the number of predicted clean instances. Specifically, it represents the probability that a defective instance in the target software will be misclassified as clean by the prediction model and is used to assess the potential risk of missing actual defects in the software. It is given by:

$$\text{FOR} = \frac{FN}{FN + TN} \tag{6}$$

An ideal system should have FOR close to 0.

4.4.2 Percent of Saved Budget (PSB)

This metric represents the fraction of total lines saved by using the model. Since the testers are more concerned about defectives, a portion of the time is saved by the

number of TNs - the number of instances where there is actually no defect and the model predicted as no defect. This speeds up the testing process and saves the testers much time. The metric is derived as:

$$\text{Percent of Saved Budget} = \frac{\sum_{i \epsilon TN} SB(LoC_i)}{\sum_{i \epsilon n_t} SB(LoC_i)} \tag{7}$$

The PSB is measured based on the additional parameter, that is lines of code. The numerator in the above equation represents the total lines of code that is present in the predicted clean modules. The denominator represents the total lines of code of a target software project, where the project consists of n_t modules. Here, we assume that each line of code represents a unit cost for verification.

4.5 Statistical Significance Test

To investigate the significance difference in the performances of the proposed optimal threshold approach and the other models, two non-parametric significance tests, such as the *one-sample Wilcoxon signed rank test* and *Cliff's delta effect size* tests, were conducted. The results of the significance tests are provided using three performance measures, such as FOR and PSB, and the difference between the probability of detection and the probability of false alarm. The *one-sample Wilcoxon signed rank test* is effective over the parametric significance tests because it does not consider the distribution of the defect data while evaluating the hypothesis testing. The null and the alternative hypothesis for *one-sample Wilcoxon signed rank test* is given below:
H_0: The average performance of the other models is equal to the performance of the optimal threshold approach.
H_1: The average performance of the other models differs from the performance of the optimal threshold approach.
For this, the customary threshold value is taken as 0.05. The null hypothesis is rejected when the p-value is less than 0.05 significance level Demšar (2006).

In addition to *one-sample Wilcoxon signed rank test*, to know the amount of difference between the proposed optimal threshold approach and the other models, such as the base-line probabilistic classifier and the counterfactual approach, we conduct another non-parametric effect size test (EST) called the Cliff's delta. This measure provides four levels of effectiveness of the optimal threshold approach on the other models over the target projects. These levels are given in table 2. The larger value of Cliff's delta indicates the greater effect between the models.

Table 2: Cliff's delta effect size levels Cliff (1993)

S.No	$\lvert\delta\rvert$	Effectiveness Category
1	$0.000 \leq \lvert\delta\rvert < 0.147$	Negligible
2	$0.147 \leq \lvert\delta\rvert < 0.330$	Small
3	$0.330 \leq \lvert\delta\rvert < 0.474$	Medium
4	$0.474 \leq \lvert\delta\rvert \leq 1.000$	Strong

Test set	Optimal Threshold	The Difference Between Probability of Detection and Probability of False Alarm			FOR			PSB		
3-11		Initial	Counterfactual Approach [TP(1) + TN(2)]	Optimal Threshold Approach	Initial	Counterfactual Approach [TP(1) + TN(2)]	Optimal Threshold Approach	Initial	Counterfactual Approach [TP(1) + TN(2)]	Optimal Threshold Approach
Ant	0.4	0.171	0.286	0.314	0.192	0.166	0.154	66.77	63.26	56.95
Camel	0.4	0.032	0.021	0.111	0.187	0.191	0.169	67.65	65.2	49.5
Forrest	0.3	0	0	0.333	0.062	0.062	0	95.86	95.86	28.93
Ivy	0.4	0.332	0.316	0.363	0.081	0.076	0.065	72.23	68.65	59.43
Jedit	0.8	0.284	0.284	0.337	0.015	0.014	0.015	73.47	60.35	93.08
Log4j	0.3	0.032	0.053	0.1	0.92	0.918	0.897	6.84	6.84	1.42
Lucene	0.3	0.104	0.116	0.192	0.571	0.566	0.44	39.39	38.18	16.94
Poi	0.3	0.105	0.085	0.37	0.609	0.614	0.265	26.96	26.81	10.23
Synapse	0.3	0.174	0.237	0.343	0.284	0.273	0.121	55.14	52.26	24.37
Tomcat	0.5	0.414	0.471	0.414	0.048	0.035	0.049	67.99	54.12	68.14
Velocity	0.4	0.102	0.089	0.182	0.321	0.319	0.291	61.66	60.49	54.55
Xalan	0.2	0.109	0.138	0.377	0.987	0.986	0.971	1.1	1.1	0.62
Xerces	0.4	0.081	0.106	0.126	0.726	0.721	0.715	17.72	17.72	16.32
Average	0.3538	0.1492	0.1694	**0.2740**	0.3848	0.3801	**0.3194**	50.2138	46.9877	**36.96**
Cliff's Delta		**0.6036**	**0.5266**	-	**0.1538**	**0.1361**	**-**	**0.3136**	**0.2544**	**-**
Wilcoxon Test		**0.0096**	**0.0222**	-	0.0522	0.0579	**-**	**0.0186**	**0.0287**	**-**

Table 3: Results of Various Models using Logistic Regression Model

Test set 3-11	Optimal Threshold	The Difference Between Probability of Detection and Probability of False Alarm			FOR			PSB		
		Initial	Counterfactual Approach [TP(1) + TN(2)]	Optimal Threshold Approach	Initial	Counterfactual Approach [TP(1) + TN(2)]	Optimal Threshold Approach	Initial	Counterfactual Approach [TP(1) + TN(2)]	Optimal Threshold Approach
Ant	0.1	0.282	0.281	0.36	0.166	0.166	0.145	63.08	63.64	59.23
Camel	0.1	0.072	0.084	0.11	0.182	0.18	0.175	68.2	68.36	63.03
Forrest	0.1	0	0	0.067	0.062	0.062	0.067	95.86	95.86	87.22
Ivy	0.1	0.341	0.338	0.374	0.073	0.074	0.067	70.37	70.14	65.02
Jedit	0.9	0.238	0.22	0.261	0.016	0.016	0.015	59.91	56.58	63.01
Log4j	0.8	-0.01	-0.01	0.042	0.923	0.923	0.919	6.07	6.07	6.84
Lucene	0.1	0.129	0.119	0.149	0.562	0.564	0.554	30.4	29.73	28.49
Poi	0.1	0.114	0.113	0.142	0.606	0.606	0.597	26.8	27	26.14
Synapse	0.1	0.22	0.231	0.324	0.277	0.274	0.241	47.86	47.86	44.63
Tomcat	0.1	0.42	0.417	0.536	0.049	0.049	0.034	66.6	66.49	61.48
Velocity	0.5	0.121	0.121	0.121	0.311	0.311	0.311	53.51	53.51	53.51
Xalan	0.5	0.249	0.251	0.313	0.984	0.984	0.982	1.1	1.1	1.1
Xerces	0.1	0.138	0.131	0.159	0.713	0.715	0.707	17.72	17.72	17
Average	0.3	0.1780	0.1766	**0.2275**	0.3788	0.3788	**0.3703**	**46.7292**	46.4662	44.3615
Cliff's Delta		**0.2012**	**0.2130**	-	**0.0592**	**0.0592**	-	**0.1125**	**0.1006**	-
Wilcoxon Test		**0.0397**	**0.0369**	-	0.0817	0.0817	-	0.0645	0.0682	-

Table 4: Results of Various Models using Naïve Bayes Model

Test set 3-11	Optimal Threshold	The Difference Between Probability of Detection and Probability of False Alarm			FOR			PSB		
		Initial	Counterfactual Approach [TP(1) + TN(2)]	Optimal Threshold Approach	Initial	Counterfactual Approach [TP(1) + TN(2)]	Optimal Threshold Approach	Initial	Counterfactual Approach [TP(1) + TN(2)]	Optimal Threshold Approach
Ant	0.5	0.313	0.298	0.388	0.158	0.161	0.134	60.72	58.66	55.04
Camel	0.4	0.114	0.105	0.131	0.171	0.178	0.162	53.31	45.31	43.54
Forrest	0.4	0	0	0	0.062	0.07	0	95.86	68.12	0
Ivy	0.4	0.378	0.381	0.484	0.067	0.08	0.026	66.1	41.78	43.75
Jedit	0.8	0.136	0.135	0.136	0.017	0.02	0.017	42.45	43.75	42.45
Log4j	0.2	0.043	0.041	0.065	0.918	0.963	0.916	6.07	8.94	5.31
Lucene	0.2	0.176	0.166	0.28	0.546	0.6	0.462	30.52	19.56	18.96
Poi	0.4	0.28	0.19	0.505	0.547	0.55	0.368	23.5	20.79	17.77
Synapse	0.4	0.012	0.02	0.3	0.333	0.35	0.245	61.16	58.91	45.58
Tomcat	0.4	0.053	0.09	0.186	0.085	0.09	0.064	79.07	41.72	39.8
Velocity	0.4	0.019	0.02	0.285	0.336	0.335	0.234	59.36	36.45	29.22
Xalan	0.2	0.198	0.187	0.561	0.985	0.971	0.969	1.1	0.78	0.89
Xerces	0.2	0.192	0.188	0.452	0.697	0.746	0.479	16.52	10.96	11.25
Average	0.3769	0.147	0.140	**0.290**	0.379	**0.393**	0.340	**45.826**	35.056	27.197
Cliff's Delta	**0.4616**	**0.4734**	-	**1**	**0.1716**	-	**0.4261**	**0.2131**	-	
Wilcoxon Test	**0.0483**	**0.0427**	-	**1.92E-07**	**0.4983**	-	**0.0483**	**0.3694**	-	

Table 5: Results of Various Models using Neural Networks Model

5 Study Results

This section beriefs the obtained results in the conducted experiments. The proposed techniques are tested on PROMISE datasets. The results presented in tables 3, 4 and 5 provide insights into the performance of the models on the PROMISE datasets.

5.1 Performances using Logistic Regression

Table 3 shows the results of individual techniques using the Logistic Regression model. Initial technique refers to the results that were generated without considering the balance between the probability of detection and probability of false alarm. The Optimal Threshold column indicate the results obtained using the proposed iterative approach , we notice that the average optimal threshold is 0.35 but not 0.5.

Considering the average values row, the optimal threshold approach worked better, giving the maximum distance while reducing FOR. The counterfactual approach also gave better results than the initial values. Consider figure 1(a) where the optimal threshold is marked as 0.4. The difference at this threshold is 0.143 more than at threshold 0.5. Moreover, the FOR at the optimal threshold is less and can be seen from the figure. Similarly, for other test sets, the optimal threshold gave the maximum difference between the probability of detection and the probability of false alarm.

5.2 Performances using Naïve Bayes

Table 4 shows the results of individual techniques using the Naïve Bayes model. The average optimal threshold is 0.3, indicating that the model performed at thresholds lower than 0.5.

Similar to the Logistic Model trends, the optimal threshold approach gave the best performance for both the difference between pd and pda, and FOR metric. However, the PSB of the initial approach is greater, but the difference is not substantial.

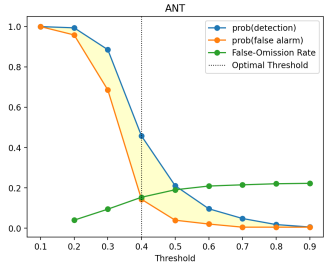
It can be seen from figure 2 that both pd and pda are wide apart, indicating that the model performed well and can identify the essence of the difference at almost all threshold values. At thresholds where the difference is negligible, the model formed well on the FOR metric by selecting the threshold with the minimum FOR value.

5.3 Performances using Neural Network

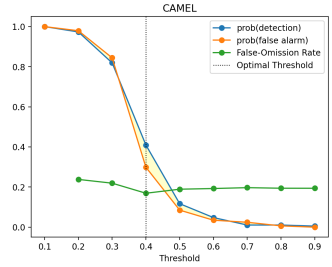
Table 4 shows the results of individual techniques using the Neural Network. The average optimal threshold is 0.38, smaller than the default 0.5.

Comparing different approaches using this model, the optimal threshold approach gave the maximum difference between the metrics compared to initial values and the counterfactual approach. However, the FOR value is the least for the counterfactual approach, showing that the model benefitted from the generated data. Considering the PSB value, both the proposed approaches have lower percentages, but when deployed, considering the risk it is not the main factor.

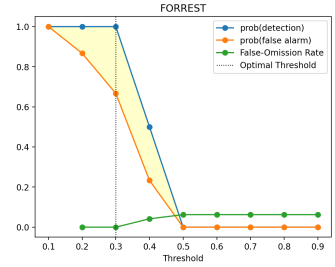
One observation from figure 3 is that for some test sets, the pfa value exceeded pd value (usually at lower thresholds). Similar to the previous models, the maximum difference is considered and can be seen from subfigures 3(a) and 3(c).



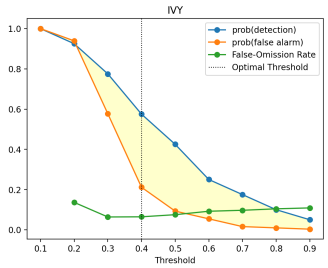
((a)) Performances on the Project *Ant-1.7*



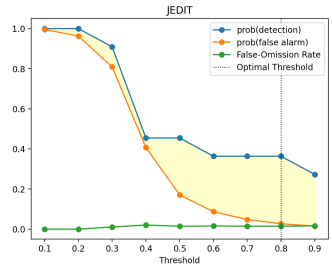
((b)) Performances on the Project *Camel-1.6*



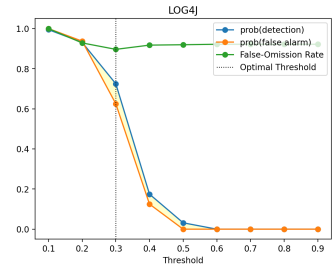
((c)) Performances on the Project *Forrest*



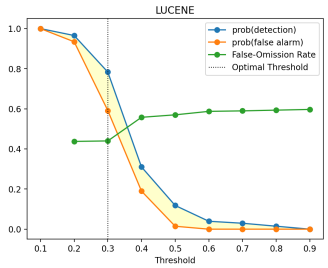
((d)) Performances on the Project *Ivy*



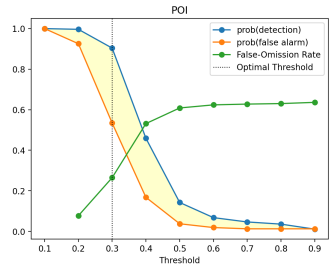
((e)) Performances on the Project *JEdit-4.3*



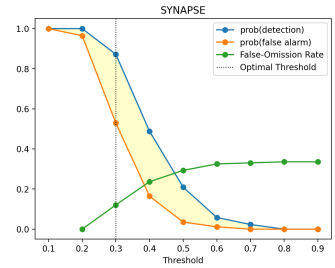
((f)) Performances on the Project *Log4j-1.2*



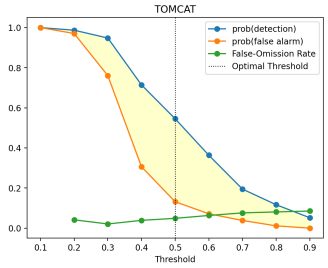
((g)) Performances on the Project *Lucene-2.4*



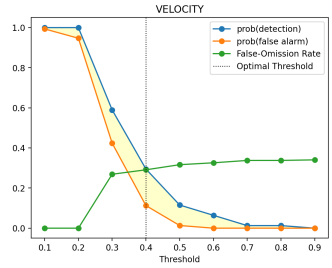
((h)) Performances on the Project *Poi-3.0*



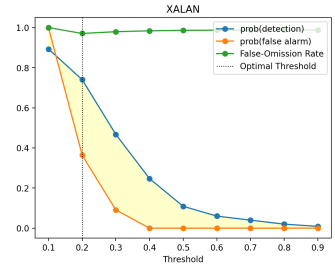
((i)) Performances on the Project *Synapse-1.2*



((j)) Performances on the Project *Tomcat*

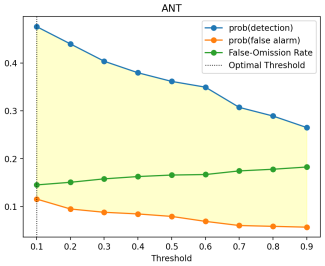


((k)) Performances on the Project *Velocity-1.6*

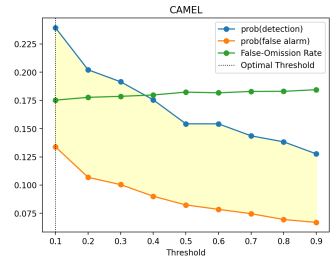


((l)) Performances on the Project *Xalan-2.7*

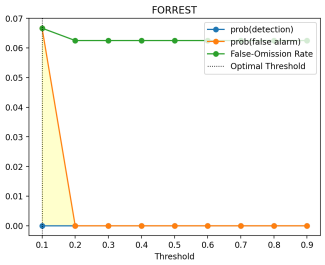
Figure 1: Results on Logistic Regression Model



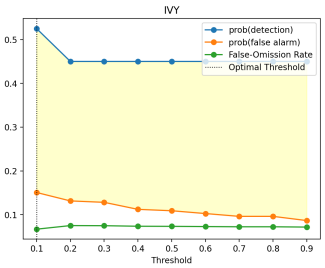
((a)) Performances on the Project *Ant-1.7*



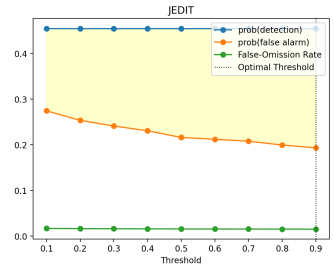
((b)) Performances on the Project *Camel-1.6*



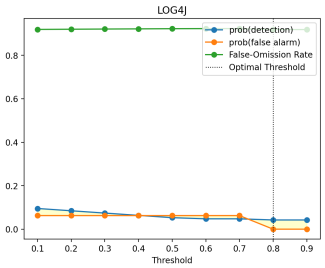
((c)) Performances on the Project *Forrest*



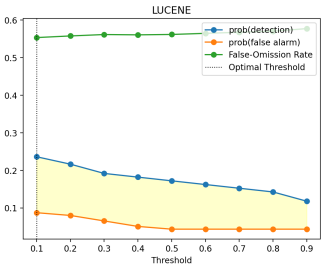
((d)) Performances on the Project *Ivy*



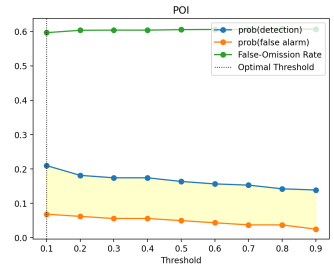
((e)) Performances on the Project *JEdit-4.3*



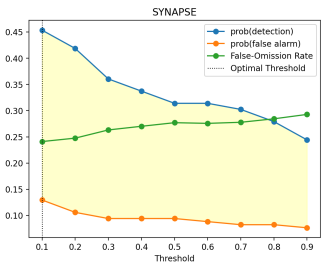
((f)) Performances on the Project *Log4j-1.2*



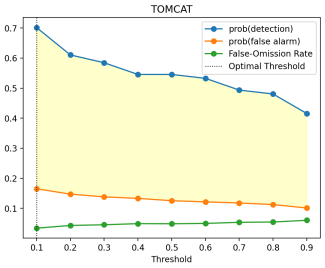
((g)) Performances on the Project *Lucene-2.4*



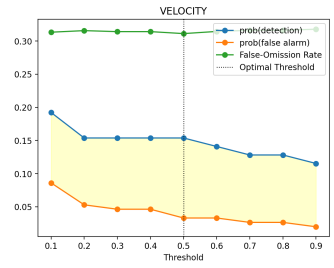
((h)) Performances on the Project *Poi-3.0*



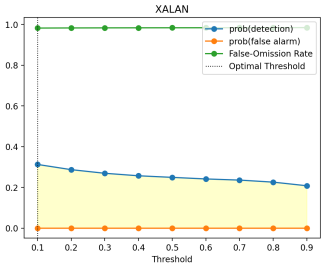
((i)) Performances on the Project *Synapse-1.2*



((j)) Performances on the Project *Tomcat*

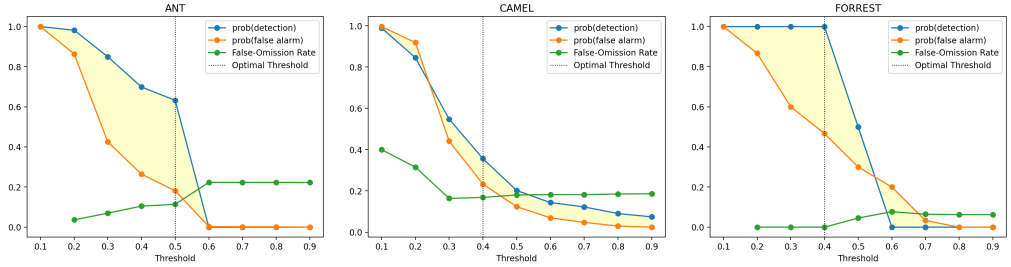


((k)) Performances on the Project *Velocity-1.6*



((l)) Performances on the Project *Xalan-2.7*

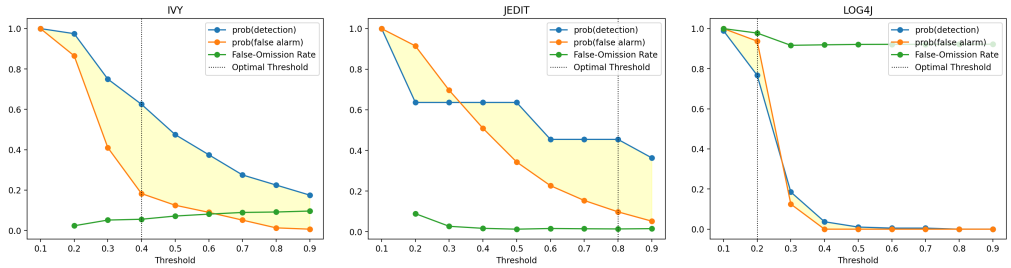
Figure 2: Results on Naïve Bayes Model



((a)) Performances on the Project *Ant-1.7*

((b)) Performances on the Project *Camel-1.6*

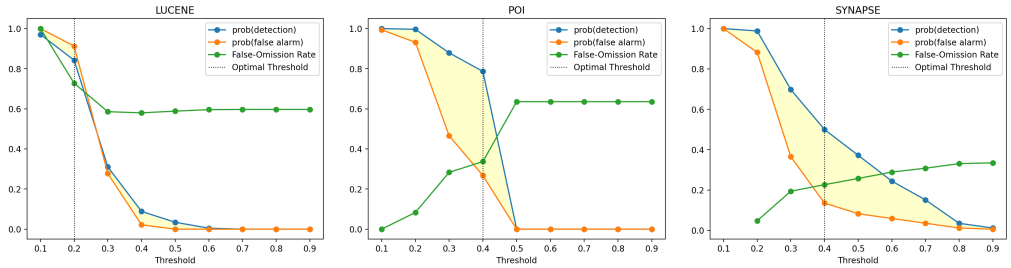
((c)) Performances on the Project *Forrest*



((d)) Performances on the Project *Ivy*

((e)) Performances on the Project *JEdit-4.3*

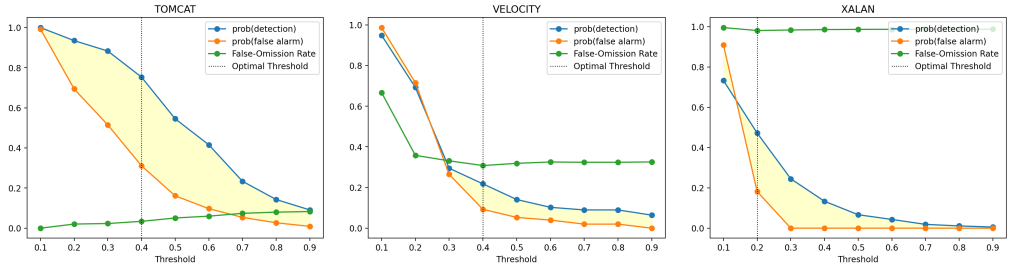
((f)) Performances on the Project *Log4j-1.2*



((g)) Performances on the Project *Lucene-2.4*

((h)) Performances on the Project *Poi-3.0*

((i)) Performances on the Project *Synapse-1.2*



((j)) Performances on the Project *Tomcat*

((k)) Performances on the Project *Velocity-1.6*

((l)) Performances on the Project *Xalan-2.7*

Figure 3: Results on Neural Networks Model

5.4 Discussion

The optimisation function aims to determine the optimal threshold for classifying defects, and the results include the threshold value, false omission rate (FOR), and the percentage of lines of code (% saved LOC). It is evident from both results that the optimal threshold is not the default 0.5. It varies according to the test set and its distribution. By dynamically determining the optimal threshold, the models achieve a balance between correctly identifying defects and minimizing false alarms. The results indicate that the optimisation function helps identify thresholds that lead to higher FOR values and greater savings in debugging efforts.

Overall, the proposed optimisation function shows varying performance across different models and test sets in the PROMISE dataset. While some models demonstrate strong defect detection capabilities and significant savings in terms of debugging efforts, others exhibit higher false omission rates and limited potential for reducing debugging efforts.

In the fine-tuning model with counterfactual explanations approach, we generated counterfactual explanations based on TP and TN as they are already correctly classified by the model and the assumption that the model has an understanding of those instances in the feature space. This allows us to generate artificial instances that are more robust and fall outside the decision boundary of the model.

The optimal threshold approach's feasibility can be seen from Figures 1 2 and 3. Consider 1(a) for a detailed analysis. As mentioned in Section 3, an ideal model will have high *prob(detection)* and low *prob(false alarm)*. From the figure, at lower thresholds, both the metrics are high and at higher thresholds, both are low, which is not desirable. To have the highest *prob(detection)* while having the lowest *prob(false alarm)*, considering the distances between the metrics, 0.4 seems to give the optimal result. We can also see that the *FOR* metric is lower than the default 0.5 at this threshold.

5.5 Threats to Validity

In this section, we discuss the limitations and threats that validate the performance of the proposed SDP approaches for maximising the difference between the probability of detection and the probability of false alarms.

5.5.1 Construct Validity

Construct validity refers to the extent to which the parameters and the metrics used in the study accurately represent the theoretical constructs they are intended to measure. In our context, ensuring construct validity is crucial for producing meaningful and reliable results.

Different projects may have varying definitions of what constitutes a defect. This can lead to inconsistencies in the data and affect the validity of the prediction models. Another threat is that focusing solely on the probability of detection and the probability of false alarms might overlook other important aspects of model performance, such as precision, recall, F1-score, and AUC. However, we utilised measures such as the FOR, and PSB, to measure the performances of the various approaches.

The features selected for the model might not fully capture the factors that influence software defects. Irrelevant or poorly chosen features can lead to biased or inaccurate predictions. Since our focus is to analyse the impact of maximising the probability of detection and the probability of false alarm, we did not conduct experiments on feature selection techniques in building the proposed approaches. A potential way to mitigate this issue is to use domain knowledge and consult with software engineering experts to select relevant features. Performing exploratory data analysis (EDA) and using statistical tests to validate the significance of each feature.

The use of counterfactual explanations to generate new data points assumes that the counterfactual instances are realistic and relevant to the actual software defects is the potential threat that validates the performance of the implemented EXI model. Validating the generated counterfactual data points with domain experts is a way to ensure they represent plausible scenarios. Additionally, experimenting to compare the performance of models trained with and without counterfactual data to assess the added value.

In our experiments, we adjusted the threshold value to balance Pd and Pf however, it might result in a model that performs well on these metrics but poorly on others. Experimenting with different threshold values using different measures to report the changes is a way to mitigate this challenge but, incorporating the vast results may be infeasible.

5.5.2 Internal Validity

Internal validity refers to whether the observed performances of the SDP approaches are due to the experimental treatments and not other factors.

This work utilises 36 software project information to validate the performance of the proposed approaches. The performance of the proposed approaches depends largely on the quality of the utilised data and the labelling of the corresponding data points Pachouly et al. (2022). Though the utilised data is publicly available, there are issues with the quality Bhandari et al. (2023); Pachouly et al. (2022); Bowes et al. (2018), mainly due to the level of the experts who are testing and labelling the defect-prone and clean modules. Another reason is, since generating the universal set of test cases to test individual modules, it is impractical to label some modules as defective Bhutamapuram and Sadam (2022); Pressman (2005); Fenton and Neil (1999); Fenton and Ohlsson (2000). Hence, mislabeling or inconsistent labeling can introduce bias in developing the models. However, this work does not explore the methods to improve the quality of the data, hence, we may observe the improved performances if the the quality of the data is improved. Other variables not included in the model might influence the defect prediction outcomes. For example, developer experience or code complexity.

5.5.3 External Validity

This section discusses the generalisability of the approaches proposed in this work to different projects. This is because, the datasets used for training and testing might not be representative of all types of software projects, limiting the generalizability of the findings. Hence, our approach may produce variability in the predictions when we use

a diverse set of datasets from different domains, project types, and development environments. To assert the proposed models are good at improving the performance, we need to conduct cross-project validation studies. The other threat that diminishes the performance of the proposed approach and the robustness of the results is the sample size that we used in the empirical analysis. Though we implemented the approaches in the cross-project setting, some projects may not contain sufficient data to test the potential of the approaches. Some examples include the projects such as *Ant-1.3, Ant-1.4, Log4j-1,0, Log4j-1.1, Lucene-2.0, Synapse-1.0*, and *Velocity-1.4* contain the data less than 200 data points. However, none of the projects mentioned above were used to test the trained models.

6 Conclusion

In this research work, we woked on a new objective that maximises the difference between the probability of detection and the probability of false alarms particularly in the context of software defect prediction. To solve the objective, we propose 2 techniques: exploring the optimal threshold and fine-tuning the model by generating new instances from counterfactual explanations. The proposed methodology can be applied to to any probabilistic model that outputs a probability score of an instance. The results of our experiments demonstrate the effectiveness of the proposed method in achieving a better balance between false positives and false negatives, thereby, adhereing to industry needs. It is also evident that the optimal decision threshold is not 0.5 but may be higher or lower based on the data points and the model trained. We also show that though counterfactuals improve the performance as compared to the base results, it does not give the best results as the model itself is not 100% accurate in generating counterfactuals that exactly fall into the desired classes. Overall, the proposed methods can contribute to the development of more accurate and reliable classifiers for real-world applications.

Acknowledgements

We, the authors declare that, the research presented in this paper was conducted independently, and the authors have no financial, professional, or personal relationships that could be perceived as influencing the results or conclusions reported in this work. Also, we acknowledge that, we make use of AI tools to improve the scientific writing.

7 Authors' Contribution

Mr. Nitin Sai Bommi

1. Conceptualization
2. Survey
3. The research gap
4. Objective definition
5. Methodology
6. Data Curation
7. Experiments

8. Results Validation
9. Writing the draft of the paper

Dr. Umamaheswara Sharma B

1. Conceptualization
2. Survey
3. The research gap
4. Objective definition
5. Methodology
6. Data Curation
7. Supervision
8. Results Validation
9. Writing the draft of the paper

Dr. Atul Negi

1. Supervision
2. Writing the draft of the paper

8 Funding

The authors have received no financial support from any public sector or private sector organisations for the research, authorship, and publication of this article.

9 Data Availability

The data that support the findings of this study are obtained from the AEEEM repository, which is a publicly available repository, published by D'Ambros et al. D'Ambros et al. (2012).

10 Conflict of Interest

The authors declare that they have no conflicts of interest with all the editorial board members of the Software Quality Journal.

References

Herbold, S., Trautsch, A., Grabowski, J.: A comparative study to benchmark cross-project defect prediction approaches. IEEE Transactions on Software Engineering **44**(9), 811–833 (2017)

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. IEEE Transactions on Software Engineering (2017)

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: The impact of automated parameter optimization on defect prediction models. IEEE Transactions on Software Engineering **45**(7), 683–711 (2018)

B, U.S., Sadam, R.: How far does the predictive decision impact the software project? the cost, service time, and failure analysis from a cross-project defect prediction model. Journal of Systems and Software **195**(1), 111522 (2023) https://doi.org/10.1016/j.jss.2022.111522

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., Murphy, B.: Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 91–100. ACM, Amsterdam The Netherlands (2009)

Ma, Y., Luo, G., Zeng, X., Chen, A.: Transfer learning for cross-company software defect prediction. Information and Software Technology **54**(3), 248–256 (2012)

Mizuno, O., Hirata, Y.: A cross-project evaluation of text-based fault-prone module prediction. In: 2014 6th International Workshop on Empirical Software Engineering in Practice, pp. 43–48 (2014). IEEE

Xia, X., Lo, D., Pan, S.J., Nagappan, N., Wang, X.: Hydra: Massively compositional model for cross-project defect prediction. IEEE Transactions on software Engineering **42**(10), 977–998 (2016)

Wu, F., Jing, X.-Y., Sun, Y., Sun, J., Huang, L., Cui, F., Sun, Y.: Cross-project and within-project semi-supervised software defect prediction: A unified approach. IEEE Transactions on Reliability **67**(2), 581–597 (2018)

Xu, Z., Pang, S., Zhang, T., Luo, X.-P., Liu, J., Tang, Y.-T., Yu, X., Xue, L.: Cross project defect prediction via balanced distribution adaptation based transfer learning. Journal of Computer Science and Technology **34**(5), 1039–1062 (2019)

Jin, C.: Cross-project software defect prediction based on domain adaptation learning and optimization. Expert Systems with Applications **171**, 114637 (2021)

Tantithamthavorn, C.: Towards a better understanding of the impact of experimental components on defect prediction modelling. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 867–870 (2016)

Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering **34**(4), 485–496 (2008)

Tantithamthavorn, C., Hassan, A.E.: An experience report on defect modelling in practice: Pitfalls and challenges. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, pp. 286–295 (2018)

Shalev-Shwartz, S., Ben-David, S.: Understanding Machine Learning: From Theory to Algorithms. Cambridge university press, 40 W. 20 St. New York, NYUnited States

(2014)

Bishop, C.M., Nasrabadi, N.M.: Pattern Recognition and Machine Learning. 4, vol. 4. Springer, Springer New York, NY (2006)

Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. Journal of Machine Learning Research **3**, 1157–1182 (2003)

James, G., Witten, D., Hastie, T., Tibshirani, R.: An Introduction to Statistical Learning vol. 112. Springer, Springer New York, NY (2013)

Morasca, S., Lavazza, L.: On the assessment of software defect prediction models via roc curves. Empirical Software Engineering **25**, 3977–4019 (2020)

Lyu, M.R., *et al.*: Handbook of Software Reliability Engineering vol. 222. IEEE computer society press, Los Alamitos, California (1996)

Molnar, C.: Interpretable Machine Learning. Lulu. com, ??? (2020)

Tantithamthavorn, C., Cito, J., Hemmati, H., Chandra, S.: Explainable ai for se: Challenges and future directions. IEEE Software **40**(3), 29–33 (2023)

Tosun, A., Bener, A.: Reducing false alarms in software defect prediction by decision threshold optimization. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 477–480 (2009). IEEE

Tantithamthavorn, C.K., Jiarpakdee, J.: Explainable ai for software engineering. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1–2 (2021). IEEE

Bommi, N.S., Negi, A.: A standard baseline for software defect prediction: using machine learning and explainable ai. In: 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 1798–1803 (2023). IEEE

Khanan, C., Luewichana, W., Pruktharathikoon, K., Jiarpakdee, J., Tantithamthavorn, C., Choetkiertikul, M., Ragkhitwetsagul, C., Sunetnanta, T.: Jitbot: an explainable just-in-time defect prediction bot. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 1336–1339 (2020)

Jiarpakdee, J., Tantithamthavorn, C.K., Grundy, J.: Practitioners' perceptions of the goals and visual explanations of defect prediction models. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 432–443 (2021). IEEE

Jiarpakdee, J., Tantithamthavorn, C.K., Dam, H.K., Grundy, J.: An empirical study of model-agnostic techniques for defect prediction models. IEEE Transactions on Software Engineering **48**(1), 166–185 (2020)

Aleithan, R.: Explainable just-in-time bug prediction: Are we there yet? In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 129–131 (2021). IEEE

Sayyad Shirabad, J., Menzies, T.J.: The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada (2005). http://promise.site.uottawa.ca/SERepository

Ramezankhani, A., Pournik, O., Shahrabi, J., Azizi, F., Hadaegh, F., Khalili, D.: The impact of oversampling with smote on the performance of 3 classifiers in prediction of type 2 diabetes. Medical decision making **36**(1), 137–144 (2016)

Blagus, R., Lusa, L.: Smote for high-dimensional class-imbalanced data. BMC bioinformatics **14**, 1–16 (2013)

Agrawal, A., Menzies, T.: Is" better data" better than" better data miners"? on the benefits of tuning smote for defect prediction. In: Proceedings of the 40th International Conference on Software Engineering, pp. 1050–1061 (2018)

Bhutamapuram, U.S., Sadam, R.: With-in-project defect prediction using bootstrap aggregation based diverse ensemble learning technique. Journal of King Saud University - Computer and Information Sciences **34**(10, Part A), 8675–8691 (2022)

Herbold, S.: Training data selection for cross-project defect prediction. In: Proceedings of the 9th International Conference on Predictive Models in Software Engineering, pp. 1–10 (2013)

Nam, J., Fu, W., Kim, S., Menzies, T., Tan, L.: Heterogeneous defect prediction. IEEE Transactions on Software Engineering **44**(9), 874–896 (2017)

Demšar, J.: Statistical comparisons of classifiers over multiple data sets. The Journal of Machine learning research **7**, 1–30 (2006)

Cliff, N.: Dominance statistics: Ordinal analyses to answer ordinal questions. Psychological bulletin **114**(3), 494 (1993)

Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., Abraham, A.: A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. Engineering Applications of Artificial Intelligence **111**, 104773 (2022)

Bhandari, K., Kumar, K., Sangal, A.L.: Data quality issues in software fault prediction: a systematic literature review. Artificial Intelligence Review **56**(8), 7839–7908 (2023)

Bowes, D., Hall, T., Petrić, J.: Software defect prediction: do different classifiers find the same defects? Software Quality Journal **26**(2), 525–552 (2018)

Pressman, R.S.: Software Engineering: a Practitioner's Approach. Palgrave macmillan, 1221 Avenue of the Americas, New York (2005)

Fenton, N.E., Neil, M.: A critique of software defect prediction models. IEEE Transactions on software engineering **25**(5), 675–689 (1999)

Fenton, N.E., Ohlsson, N.: Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software engineering **26**(8), 797–814 (2000)

D'Ambros, M., Lanza, M., Robbes, R.: Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Software Engineering **17**(4-5), 531–577 (2012)