University of Sindh Journal of Information and Communication Technology (USJICT)
Volume 9, Issue 1, 2025

ISSN-E: 2523-1235, ISSN-P: 2521-5582
Website: http://sujo.usindh.edu.pk/index.php/USJICT/
© Published by University of Sindh, Jamshoro

A Ranking-based Approach for Effort-aware Software Defect Prediction

Areeba Zarnab[1], Muhammad Summair Raza[1], Muhammad Ramzan[1*], Mahwish Ilyas[2],
[1]Department of Software Engineering, University of Sargodha, Punjab, Pakistan (email: areebazarnab2001@gmail.com; mramzansf@gmail.com, summair.raza@uos.edu.pk)
[2] Department of Computer Science and Information Technology, University of Rasul, Mandi Bahauddin, Punjab, Pakistan (email: mahwishilyas@gmail.com)

Abstract: Software defect prediction is an important aspect of software quality assurance, as early prediction of potential defects enables better resource allocation and ensures high-quality software. Effort-aware defect prediction enhanced the testing and maintenance by identifying the modules that maximize defect detection with minimum effort. Traditional approaches use probability and defect density to rank the modules using linear regression and different other models. In this approach, we have used the Random Forest Regressor to directly predict the defects per module. Each module is assigned with a custom score to balance the two objectives: firstly, predicting the modules with maximum number of defects and secondly minimizing the required effort. We have used Particle Swarm Optimization (PSO) to optimize the scoring function with respect to effort-aware metrics including Percentage of Bugs Found at 20% effort (PofB@20%), Initial False Alarms (IFA), and Probability of Optimal Performance (Popt) in order to ensure the early defect detection and near-ideal module ranking. After the initial ranking, a defect-aware re-ranking strategy was applied to the top 20% of Lines of Code (LOC) modules by replacing them with better candidates from the remaining modules. Experimental results demonstrate that the proposed approach outperforms baseline methods, achieving a higher PofB@20% (0.402), a lower IFA (5.1), and a better Popt (0.756). The findings indicate that ranking modules based on predicted defects and inspection effort effectively helps testers detect more faults with reduced effort, confirming the superiority of the PSO-optimized ranking methodology over traditional approaches.

Keywords: Effort-aware defect prediction; Random-forest regression; Software quality assurance; Particle Swarm Optimization

I. Introduction

Software engineering is a systematic and disciplined approach that applies engineering principles throughout the software development process to ensure high-quality, reliable, and maintainable software creation. Modern software keeps getting complex day by day. Software Development Life Cycle (SDLC) is a structured and iterative framework that helps teams to follow a clear path from idea to final product [1][2] and helps to manage the complexity and quality product. Software Quality Assurance (SQA) plays a very important role in ensuring that quality practices are applied throughout the software development life cycle [3]. Testing is an important component of SQA, aiming to detect bugs as early as possible, but in reality, the testing of every single module is not possible [4]. That's where the concept of software defect prediction techniques come in; by using historical software metrics and machine learning models, teams can focus their efforts on the parts of the system most likely to contain bugs [5]. Traditional defect prediction methods used classification algorithms, which assign modules into defective or non-defective categories. But these methods do not consider the cost or effort required to inspect or test each module [6]. Attributes like LOC, complexity, and how often code has changed are used to estimate effort [7]. In this way, Effort Aware Software Defect Prediction (EASDP) helps testers prioritize modules that are both risky and efficient to review, which gives us better results with fewer resources. Several studies have proposed ranking-based approaches for EASDP, where modules are ordered according to composite scores combining predicted defectiveness and estimated effort. Focus on static probability or defect density; they do not optimize for early defect detection or dynamically respond when the top-ranked modules are ineffective. To address these limitations, this research proposes a new ranking-based approach for effort-aware defect prediction. The method does not rely on binary classification, which labels modules as defective or not. Instead, it predicts the number of bugs per module and ranks them based on a priority score that integrates predicted bug counts, effort, and other relevant factors. A hybrid optimization strategy, employing Random Forest regression for defect prediction and PSO for tuning the scoring function, is used. The main contributions of this research are as follows:

- Random Forest Regressor is employed to predict the exact no of bugs in each software module rather than just classifying modules as defective pr non-defective. This approach provides more detailed information for ranking.
- We define a scoring function that integrates predicted defect counts with the modules' LOC, allowing the method to prioritize modules that are both critical and

efficient to inspect. The parameters of the scoring function are tuned using Particle Swarm Optimization guided by a multi-objective cost function combining PofB@20%, IFA, and Popt.

We evaluated the approach across multiple project versions and compared it with state-of-the-art methods such as EALTR [8], EALR [7], CBS+ [9], and EASC [10]. While PSO-based tuning for the scoring function has appeared in prior work, our contribution lies in integrating defect count prediction using a Random Forest Regressor, cost-aware scoring PSO, PSO-based parameter tuning, and re-ranking into a single effort-aware ranking framework. Preliminary experiments show that this combination improves the prioritization of software modules under limited inspection effort.

II. Related work

This section reviews research on effort-aware software defect prediction. Many prior studies rely on datasets such as NASA, RELINK, and SOFTLAB, which contain only class labels rather than actual defect counts [11]. Replacing bug counts with class labels can reduce detected defects and increase false alarms when using LOC as an effort. Therefore, including bug counts in datasets is important for building effective EASDP models. A regression–based defect prediction approach with feature selection and feature optimization [12] is used to enhance accuracy. The performance of the regression-based defect prediction approach was evaluated by comparing it with other models in terms of accuracy, precision recall on the CMI and promise datasets. The results showed that the SVM classifier shows the highest accuracy of 99%. Regression-based models have served as a foundational strategy for many EASDP approaches. These models predict defect densities or probabilities using conventional software metrics such as Lines of Code, Cyclomatic Complexity, Coupling, and Cohesion. The values of these metrics are then used to inspect the modules. In the study [8], the authors developed a method for effort aware defect prediction by using the linear regression. The coefficient of linear regression was generated using a differential evolution algorithm. Further, a reranking strategy was proposed to reduce false alarms. Results were then evaluated using elven (110 different datasets. Another significant work [13] introduced improved effort-aware metrics and addressed some common issues of EASDP evaluation. The authors identified the inconsistencies in the use of PofB@20% metrics and proposed a normalized variant based on ranking by predicted defect density. The normalization improved the consistency between classifier outputs and actual defect-prone modules. In [14], author presented a change-level defect prediction method

called Just-In-Time (JIT) defect prediction to identify the risky code. The proposed approach used fourteen (14) change-related metrics, such as lines added, number of modified files, and developer experience, and then applied the logistic regression to predict whether a change introduces a defect. They used Effort-aware metrics i.e., PofB@20%, IFA, and Popt to evaluate the performance. While JIT showed some promising results when there were no restrictions on inspection effort, it performed poorly in case of less amount of code. In such constrained-effort scenarios, the Effort-Aware Linear Regression (EALR) model outperformed JIT by providing better rankings of defect-prone modules relative to the effort required, making it more suitable for practical settings where testing resources are limited. Optimization algorithms have been applied to tune model parameters or scoring functions to enhance the defect detection performance under limited inspection budgets. In [15], the authors proposed a multi-objective effort-aware defect prediction approach based on NSGA-II, named MOOAC for EASDP. The goal of this approach is to maximize the PofB@20% metric and minimize the PMI@20% when inspecting the top 20% of the line of code. The MOOAC approach combines random forest classification with logistic regression in a multi-objective optimization framework. In [16], the authors proposed CoreBug, an effort-aware prediction strategy that captured nine different kinds of coupling by using a Weighted Directed Class Dependency Network (WDCDN). Secondly, they introduced a generalized k-core decomposition to give every class a coreness score, then they combined each class's coreness with its relative risk from a logistic regression model. The author evaluated the performance of this proposed methodology on 18 Java projects. The results showed that CoreBug outperformed all existing methods. Class imbalance is the most common problem in software defect prediction. The hybrid framework, SAGA [17], was introduced to solve the imbalance problem in software defect prediction. Learning-to-Rank (L2R) approaches are designed to optimize ranking quality directly, rather than treating defect prediction as a traditional classification or regression task. These methods are particularly useful when the primary goal is to prioritize defect-prone modules under budget constraints. Previous studies preferred an unsupervised model for ranking because a supervised model performed poorly compared to an unsupervised model. Due to this limitation, authors in [18] proposed a semi-supervised model for just-in-time (JIT) defect prediction known as Effort-Aware Tri-Training (EATT), which used a greedy strategy. They evaluate the performance of EATT on six open-source projects and compare it with other supervised and unsupervised models. The results showed that EATT achieved high classification accuracy as a supervised model. A Differential evolution (DE)-based supervised method for effort-aware just-in-time software defect prediction (JIT-SDP) was proposed in [19]. The goal of JIT-SDP is to identify defective software changes while optimizing limited software testing resources. They introduce a novel metric called Density-Percentile-Average (DPA). They used logistic regression model for prediction but instead of using fixed weight they used differential evolution algorithm to find the best weights that gives the highest DPA. They compared their method with four other supervised and four unsupervised methods in cross-validation, cross-project-validation and timewise-cross-validation scenarios Effort-aware Defect Prediction was investigated to establish a stable ranking of L2R algorithms, prompted by the inconsistent findings in existing literature

[20]. So, under cross-release and cross-project settings, this study used 34 algorithms on 49 datasets for effort-aware defect prediction and measured the performance of these algorithms using 7 module-based and 7 loc-based metrics. The results showed that the random forest regressor performs best under a cross-release setting, and linear regression performs the best under a cross-project setting. To evaluate the effect of feature selection techniques [21] on EASDP, the researchers tested 24 feature selection algorithms with 10 classifiers embedded in the state-of-the-art EASDP model CBS+. They run these models on the 41 promise datasets and use six evaluation metrics for comprehensive performance assessment. The study found that different feature selection methods varied in effectiveness across classifiers and datasets for Effort-Aware Defect Prediction. Wrapper-based methods with forward search performed best, while the Correlation-based method (CorBF) from CBDP did not work well for EASDP. The features selected varied depending on the method and dataset. Optimal performance was achieved using AdaBoost, Deep Forest, Random Forest, or XGBoost as base classifiers in the EASDP model (CBS+). Therefore, the study recommended employing XGBoost with forward search in CBS+ to improve EASDP performance. CPDP in EASDP involves training models on one or more source projects and applying them to a different target project, typically with differing feature distributions and defect profiles. In [22], authors focused on effort-aware cross-project defect prediction. They introduced a method called Effort Aware Supervised Classification (EASC), which is proposed by Ni et al [10], EASC uses all cross-project modules to train a model without considering the data distribution difference between cross-project and within-project data. The authors employed different defect density calculation strategies when comparing EASC and baseline methods. To explore effective defect density calculation strategies and methods on EASC, the authors compared four data filtering methods and five transfer learning methods with EASC, using four commonly used defect density calculation strategies. They assessed the performance of these methods on eleven promise datasets using three classification evaluation metrics and seven effort-aware metrics. They found that the CBS+ defect density calculation strategy performed best, and BDA and JDA with a K-nearest neighbor classifier identified more defective modules. In [10], the authors revisited the study by Zhou et al. [25] on cross-project defect prediction, which aims to utilize models built on source projects to predict defects in target projects. Zhou et al. introduced simple unsupervised methods, ManualDown and ManualUp, which demonstrated performance comparable or superior to complex supervised methods. They recommended using ManualDown as a baseline for Non-Effort Aware Performance Measures (NPMs) and ManualUp for Effort Aware Performance Measures (EPMs). However, Zhou et al.'s comparison was limited by the use of a small subset of NPMs, reliance on results from the primary literature, and lack of evaluation of advanced EPMs addressing developer fatigue and context switches. So, based on these limitations, this study proposed an improved supervised method, EASC, to enhance Cross Project Defect Prediction (CPDP). They evaluated supervised CPDP methods against the unsupervised baselines, considering both NPMs and EPMs on 82 projects using 11 performance measures demonstrated that EASC achieved comparable performance to ManualDown for NPMs and significantly outperformed ManualUp for EPMs, with large improvements validated by Cliff's delta.

TABLE I. COMPARATIVE ANALYSIS OF EXISTING LITERATURE

Paper	Methodology	Strength	Gaps
Bug numbers matter: An empirical study of effort-aware defect prediction using class labels versus bug numbers [11]	Empirical study comparing class labels vs bug number, based on EASDP; uses ranking evaluation (PofB@20%, IFA)	Highlights the importance of bug counts for accurate effort-aware defect prediction, strong empirical evidence, and motivates regression-based ranking	Does not propose a prediction model; no optimization (PSO); no re-ranking strategy; limited to analysis of existing dataset
Enhancement of Defect Prediction Using Regression Learning Method (RLM-DP) [12]	Regression-based defect prediction with FO/FS; evaluated on CM1 & PROMISE	Predicts total defects; improves feature selection; high accuracy	No ranking; no effort-aware scoring; limited metrics; may not generalize
Improving classifier-based effort-aware software defect prediction by reducing ranking errors [23]	Proposed EA-Z ranking score strategy for classifier-based effort-aware defect prediction;	Shows improved Recall@20% and Popt; works well with imbalanced ensemble learners	Limited to classifier-based EA methods; does not explore regression-based approaches; only evaluates selected ranking strategies
Enhancing software defect prediction models using metaheuristics with a learning to rank approach [24]	Machine learning–based defect prediction using 5 models; hyperparameters optimized with 5 metaheuristic techniques	Improved prediction accuracy (13% increase); average FPA of 0.84; considers defect presence, number, and density	Focuses on hyperparameter optimization rather than ranking or effort-aware strategies
Improving effort-aware defect prediction by directly learning to rank software modules [8]	EALTR	Re-ranking strategy, Utilization of a machine learning model, Effort-aware model optimization	Limited comparison Only a Single regression model is used for ranking
On effort-aware metrics for defect prediction [13]	NPofBs	Enhanced realism Promote the practical adoption of prediction models in the industry	Focus on single EAMs Need for a more comprehensive exploration of EAMs, their trends, and normalization techniques.

Revisiting, revisiting Supervised methods for effort-aware cross-project DP [22]	compared four data filtering methods and five transfer learning methods with EASC	Multiple evaluation metrics are used Employs a non-parametric statistical test	Efficiency is missing Utilized the default hyperparameter setting The flexible adjustment of the threshold is missing
Effort-aware semi-supervised just-in-time DP [18]	EATT	Semi-supervised approach Effort-aware consideration Tri training framework	Focus on JIT defect prediction and does not address **cross-**project defect prediction. PofB@20%, EAP, and EAR are not utilized together for a comprehensive assessment.
Finding the best learning to rank algorithm for effort-aware defect prediction [20]	39 algorithms were examined across 49 datasets. to find the best learning-to-rank algorithm	Robust statistical analysis. Scott-Knott Effect Size Difference (ESD) test. Cross-release and cross-project setting	Dependency on the dataset Potential bias in algorithm selection
The impact of feature selection techniques on effort-aware defect prediction.an empirical study [21]	Comparative evaluation of feature selection methods	Comparative analysis along with Multiple evaluation metrics	Only focus on specific feature selection methods.
SAGA: A Hybrid Technique to handle Imbalanced Data in Software Defect Prediction [17]	SAGA (SMOTE + AdaSS + GA)	Solves the imbalance problem	Limited EASDP integration
CoreBug: Improving Effort-Aware Bug Prediction in Software Systems Using Generalized k-Core Decomposition in Class Dependency Networks [16]	CoreBug	Comprehensive dependency modeling	High computation cost
Revisiting Supervised and Unsupervised Methods for Effort-Aware Cross-Project Defect Prediction [10]	EASC	Superior CPDP performance	Lacks effort-aware metrics and module prioritization techniques
DEJIT: A Differential Evolution Algorithm for Effort-Aware Just-in-Time Software Defect Prediction [19]	DEJIT	Improves JIT-SDP accuracy	Limited to JIT-SDP and lacks module ranking based on effort and defects.
A multi-objective effort-aware defect prediction approach based on NSGA-II – ScienceDirect [15]	MOOAC (NSGA-II-based)	Maximizes bugs detected	Lacks integration of adjusted risk factors for ranking modules.
Revisiting Supervised and Unsupervised Methods for Effort-Aware just -in -time Defect Prediction [9]	CBS+	Higher recall, fewer false alarms fewer context switches	Uses a single classifier, static ranking, limited adaptability
A Large-Scale Empirical Study of Just-in-Time Quality Assurance [14]	Using logistic regression on 14 change-level metrics	Focuses on predicting risky changes at commit time, enabling early developer feedback	Performs poorly in constrained-effort settings; lacks strong effort-aware ranking

III. PROPOSED METHODOLOGY

A. Overview

This section presents a comprehensive methodology for an effort-aware defect prediction that integrates machine learning techniques, optimization algorithms, and a re-ranking strategy. The primary goal is to rank software modules in a way that enhances defect detection efficiency while staying within a constrained inspection effort, which is quantified using LOC. The proposed methodology starts with the preprocessing of the data. These datasets consist of different features such as static code and corresponding defect labels. In this approach, we used two versions of the datasets. The earlier versions were used to train the proposed model while the later ones were used for testing. The preprocessing steps included feature handling and normalization to ensure data quality and consistency. Then a scoring formula was used to rank the modules. The parameters α and β, were used for this purpose. These parameters were derived from PSO. Then the optimization process was applied by using a cost function that included three evaluation metrics: The percentage of bugs found in the top 20% LOC (PofB@20%), IFA, and the Popt. Once the best parameter values were obtained, modules were ranked according to these values. After this, a re-ranking strategy

was applied to ensure that the selected modules stay within the 20% LOC inspection limit. Note that this step only happens when the initial ranking does not perform well. The main purpose was to replace lower-impact modules with higher-impact ones that were excluded from the initial ranking while marinating the LOC threshold. In order to justify the proposed approach, the entire process was executed using various datasets. Figure 1 illustrates the sequential steps of our proposed methodology.

Figure 1. Proposed methodology diagram

B. Data Preprocessing and Feature Handling

We gathered datasets from multiple versions of open-source software projects, with each version available in CSV format. These datasets included module-level records, where rows corresponded to individual software components, typically classes or source files, and columns contained various static code metrics. Key features captured in these datasets included LOC, cyclomatic complexity, cohesion, coupling, inheritance depth, and method count. Alongside these metrics, a bug count was predicted for each module to reflect the number of known defects. To ensure that the learning process relied only on measurable software attributes, non-informative features such as project name and version name were removed in data processing, and during effort-aware scoring, missing values were handled, and LOC values were slightly adjusted where necessary to avoid division by zero. In this work, the defect count was explicitly treated as the prediction target and was excluded from the input feature set to prevent feature leakage. StandardScaler is used for feature scaling to stabilize the optimization of the PSO-based scoring function.

C. Model Selection

In the proposed approach, we have used Random Forest Regressor as it can handle complex, nonlinear relationships between variables. It has relatively low sensitivity to overfitting, and is easy to interpret. It also requires minimum hyper hyperparameter tuning to perform across diverse datasets. Furthermore, in order to deal with the complex relations between the features, Random Forest Regressor helps reduce overfitting, which is a common issue in various other methods. Software metric datasets typically include

various features like LOC, coupling, cohesion, and cyclomatic complexity, which normally have non-linear relation. Random Forest does not require substantial parameter tuning to capture these relations. Beyond just predictive accuracy, Random Forest offers several practical benefits that align well with the goals of this study. Its ability to find important features allowed us to see which metrics contributed most to the model's predictions and guided our feature validation process during experiments.

D. Scoring Function

In many traditional effort-aware defection or defect detection methods, modules are simply ranked based on defect probability or predicted defect count, but in this study, we first predicted the total no of bugs for each module, then used a custom scoring function that integrates both defect prediction and effort estimation. The scoring function is defined as:

$$\boldsymbol{SCORE} = \frac{\boldsymbol{\alpha.no\ of\ bugs\ Prediction}}{\mathbf{1+\beta}.\boldsymbol{LOC}} \qquad (1)$$

In this equation, prediction refers to the output of the random forest regressor, meaning the predicted no of bugs. LOC represents the line of code used as an effort. The parameters alpha and beta acts as weights that control how much influence each factor has on the score. This scoring function ensures the ranking of modules in a way that modules with high predicted defect counts but don't require a lot of effort are given higher priority in the ranking process. At the same time, very time-consuming large modules are penalized unless they have very high defect predictions. The scoring function acts as the core of the effort-aware framework, which creates a balance between effectiveness and cost and allows testers to focus on modules that give the most value. To get the best values of alpha and beta, the study used Particle Swarm Optimization, which helps automatically find the most effective parameter combination based on the evaluation metric.

E. Reranking Strategy

Before applying the reranking procedure, it is important to emphasize that the initial ranking is generated using a global scoring function and does not directly focus on the top 20% LOC inspection limit. Because of this, some large modules may appear at the top of the list due to high scores and can quickly consume the available LOC budget. To handle such cases, a reranking step is applied after selecting modules up to the 20% LOC threshold. This step checks whether better-scored modules exist outside the cutoff that can fit within the same LOC budget. If the initial ranking already places suitable modules in the top 20% LOC, no swapping is performed. The reranking step may not always lead to visible improvement; it is only applied when the initial ranking does not adequately prioritize high-value modules within the 20% LOC limit. The reranking process relies only on predicted scores and does not use actual defect information, making it suitable for practical use.

F. Optimization using PSO

PSO, a population-based stochastic optimization technique, is used to fine-tune the parameters alphaα and betaβ in the custom scoring function. PSO is a population-based optimization technique inspired by the social behavior of bird flocking. The objective function used in PSO combines three evaluation metrics, PofB@20%, IFA, and Popt, into a weighted cost function. The goal is to minimize this cost, which represents the inverse of model performance.

$$\boldsymbol{Cost} = \mathbf{0.4} * (\mathbf{1} - \boldsymbol{PofB@20\%}) + \mathbf{0.3} * (\mathbf{1} - \boldsymbol{Popt}) + \mathbf{0.3} * (\frac{\boldsymbol{IFA}}{\boldsymbol{N}}) \qquad (2)$$

Where:

PofB@20% is the proportion of bugs detected within the top 20% LOC. Popt is the optimality of the ranking. IFA represents the number of clean modules encountered before the first bug is found. The term $\frac{IFA}{N}$ normalizes IFA with respect to the total number of modules N, ensuring all metrics contribute proportionally. The weights (0.4, 0.3, 0.3) were chosen empirically to balance early bug detection and ranking efficiency. PSO iteratively updated the particle positions over multiple iterations to minimize this cost and yield the best parameter values for effective module ranking.

G. Evaluation Metrics

Quality metrics are the main factors to efficiently manage software project quality [26]. The performance of the proposed effort-aware defect prediction methodology is calculated using different metrics, PofB@20%, IFA, and Popt Recall@20% and precision@20%. The first metric, PofB@20%, measures the fraction of total bugs found within the top 20% of LOC in the ranked list. This metric directly reflects the effort-aware nature of the model. It is calculated using the formula:

$$\mathbf{PofB@20\%} = \frac{\sum_{i=1}^{k} \mathbf{bug_i}}{\sum_{j=1}^{N} \mathbf{bug_j}}, \mathbf{where} \sum_{i=1}^{k} \mathbf{LOC_i} \leq \mathbf{0.2} * \sum_{j=1}^{N} \mathbf{LOC_j} \qquad (3)$$

Here, bug_i denotes the number of bugs in module i, and the numerator accumulates bug counts from modules in the top 20% LOC. The second metric, IFA, quantifies how early the first defective module appears in the ranked list. It is defined as the number of clean (non-defective) modules encountered before the first defective one. Mathematically, it can be written as:

$$\boldsymbol{IFA} = \boldsymbol{min}\{\boldsymbol{i} \mid \boldsymbol{bug_i} > \mathbf{0}\} \qquad (4)$$

where i denotes the index of the first module with at least one bug in the sorted list. The third metric Popt, measures how close the ranking is to the optimal ranking, which places the most defective and least costly modules first, and is

calculated using the area under the cumulative lift chart. The formula is:

$$\boldsymbol{Popt} = \frac{A_{model} - A_{worst}}{A_{optimal} - A_{worst}} \quad (5)$$

where A_{model} , $A_{optimal}$, and A_{worst} represent the area under the effort–defect curve for the model, the optimal ranking, and the worst-case ranking, respectively. In addition, recall@20% and precision@20% were used to further assess the ranking quality. Precision measures the proportion of modules inspected in the top 20% LOC that are actually defective. The formula is:

$$\boldsymbol{Precision@20\%} = \frac{k}{m} \quad (6)$$

Where k represents the number of defective modules in the top 20% LOC, and m represents the total no of modules inspected in the top 20% LOC. Recall@20% quantifies the

Proportion of all defective modules in the dataset that are found within the top 20% LOC.

$$\boldsymbol{Recall@20\%} = \frac{k}{K} \quad (7)$$

k represents the number of defective modules in the top 20% LOC, and K represents the total number of defective modules in the entire dataset.

IV. Experimental Setup

In this research, an experimental setup follows a cross-version evaluation methodology. Earlier versions of a project are used to train the model, while the next version of the project is used for testing. To ensure better generalization, each run begins with an 80/20 split of the training set, where the validation part helps tune scoring parameters through the PSO algorithm. After the optimization phase, the full training dataset is used to fit a Random Forest Regressor. Predictions are then made on the test version of the software. The evaluation was performed using widely-used open-source Java projects, including Camel, Ivy, JEdit, Log4J, Lucene, POI, Synapse, Velocity, Xalan, and Xerces. And each project has multiple versions, which are stored in separate CSV files that capture the state of the software modules at that point in time. Each dataset has the same attributes, such as LOC, complexity measures, and the number of bugs. Table II shows all versions of the datasets. The Random Forest Regressor algorithm was configured with 100 estimators, a maximum depth of 7, and a minimum split of 5 to balance learning and overfitting. PSO algorithm was run with 50 particles and 10 iterations, searching for the best value of alpha within the range of 0.1 to 5.0, and beta between 0.01 and 2.0. The cost function combines three normalized components: IFA, (1 - PofB@20%), and (1 - Popt), each contributing to the overall ranking effectiveness.

TABLE II. THE DETAILS OF THE EXPERIMENTAL DATASETS

Project Version	Modules	Bugs	Faulty%
Camelv1.0	339	14	3.8%
Camel v1.2	608	522	35.5%
Camel v1.4	872	335	16.6%
Camelv1.6	965	500	19.5%
Ivy v1.1	111	233	56.8%
Ivy v1.4	241	18	6.6%
Ivy v2.0	352	56	11.4%
JEdit v3.2	372	382	33.1%
JEdit v4.0	306	226	24.5%
JEdit v4.1	312	217	25.3%
JEdit v4.3	492	12	2.2%
Log4j v1.0	135	61	25.2%
Log4j v1.1	109	86	33.9%
Log4j v1.2	205	498	92.2%
Lucene v2.0	195	268	46.7%
Lucene v2.2	247	414	58.3%
Lucene v2.4	340	632	59.7%
POI v1.5	237	342	59.5%
POI v2.0	314	39	11.8%
POI v2.5	385	496	64.4%
POI v3.0	442	500	63.6%
Synapse v1.0	157	21	10.2%
Synapse v1.1	222	99	27.0%
Synapse v1.2	256	145	33.6%
Velocity v1.4	196	210	75.0%
Velocity v1.5	214	331	66.4%
Velocity v1.6	229	190	34.1%
Xalan v2.4	723	156	15.2%
Xalan v2.5	803	531	48.2%
Xalan v2.6	885	625	46.4%
Xalan v2.7	909	1213	98.8%
Xerces v1.2	440	115	16.1%
Xerces v1.3	453	193	15.2%
Xerces v1.4	588	1596	74.3%
Xerces v(init)	162	167	47.5%

V. Results and Discussion

In the results and discussion section, we compared the performance of our proposed methodology with other baseline methods: EALTR, CBS+, EASC, and EALR. The experiments involved 25 project version pairs from the PROMISE repository datasets. We used different metrics. Table III shows the PofB@20% values for the cross-version evaluations. The percentage of Bugs found in the top 20% of code (PofB@20%) measures the fraction of total bugs found within the top 20% of LOC in the ranked list.

TABLE III. THE POFB@20% VALUE ON EACH CROSS-VERSION EXPERIMENT WHEN INSPECTING THE TOP 20%

Train	Test	Proposed Methodology	EALTR	EALR	CBS+	EASC
Camel v1.0	Camel v1.2	0.255	0.21	0.32	0.421296	0.2222

Camel v1.2	Camel v1.4	0.472	0.34	0.42	0.333333	0.5241
Camel v1.4	Camel v1.6	0.362	0.27	0.32	0.308511	0.25
Ivy v1.1	Ivy v1.4	0.278	0.22	0.38	0.3125	0.375
Ivy v1.4	Ivy v2.0	0.161	0.34	0.25	0.275	0.175
JEdit v3.2	JEdit v4.0	0.42	0.24	0.45	0.466667	0.4667
JEdit v4.0	JEdit v4.1	0.479	0.46	0.44	0.443038	0.3671
JEdit v4.1	JEdit v4.3	0.75	0.17	0.45	0.090909	0.1818
Log4j v1.0	Log4j v1.1	0.407	0.44	0.32	0.324324	0.3514
Log4j v1.1	Log4j v1.2	0.261	0.09	0.20	0.21164	0.1534
Lucene v2.0	Lucene v2.2	0.401	0.36	0.48	0.326389	0.3472
Lucene v2.2	Lucene v2.4	0.456	0.24	0.51	0.492611	0.4335
POI v1.5	POI v2.0	0.385	0.36	0.27	0.27027	0.2703
POI v2.0	POI v2.5	0.228	0.41	0.23	0.185484	0.1089
POI v2.5	POI v3.0	0.378	0.40	0.53	0.476868	0.4698
Synapsev1.0	Synapsev1.1	0.253	0.25	0.22	0.216667	0.2167
Synapse v1.1	Synapse v1.2	0.331	0.29	0.26	0.267442	0.1977
Ve(ity v1.4	Velocity v1.5	0.58	0.56	0.69	0.683099	0.6901
Velocity v1.5	Velocity v1.6	0.532	0.42	0.63	0.551282	0.5641
Xalan v2.4	Xalan v2.5	0.254	0.52	0.42	0.276486	0.1111
Xalan v2.5	Xalan v2.6	0.493	0.47	0.49	0.40146	0.4331
Xalan v2.6	Xalan v2.7	0.584	0.62	0.64	0.307606	0.3118
Xerces v1.2	Xerces v1.3	0.472	0.47	0.54	0.449275	0.1594
Xerces v1.3	Xerces v1.4	0.37	0.40	0.49	0.328638	0.0984
Xerces v1.4	Xerces v(init)	0.485	0.75	0.83	0.342466	0.3766

Table III presents the PofB@20% results, which illustrate the overall effectiveness of the proposed methodology across different software projects. In JEdit v4.1-v4.3, the proposed model achieved 0.75 values, which is greater than other baseline models. This indicates that the method successfully prioritized defective modules. Similarly, Xerces v1.4-v(init) showed strong performance with a score of 0.485, which outperformed both CBS+ and EASC. For Velocity, project (v1.4-v1.5 and v1.5-v1.6), the proposed method maintained competitive performance. A noticeable performance drop is observed on the Ivy v1.4 dataset. This behavior can be attributed to the characteristics of the dataset itself. Ivy v1.4 contains a relatively small number of defective modules, and the size of the modules varies significantly in terms of LOC. Such conditions make effort-aware ranking more challenging, as fewer defective files limit early defect discovery, while large variations in LOC reduce the effectiveness of balancing predicted defects with inspection effort. Since the proposed approach prioritizes modules under a fixed LOC budget, these dataset-specific properties can negatively influence ranking performance.

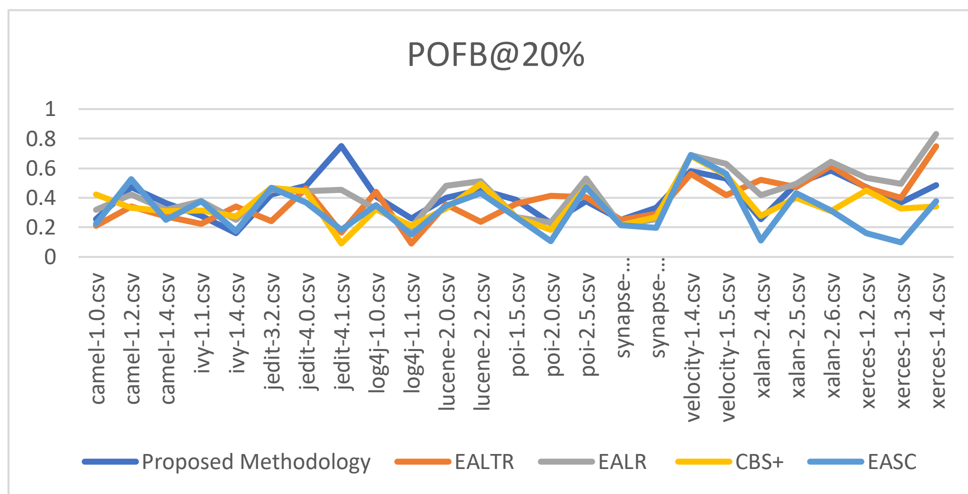

Figure 2. Comparison of PofB@20% performance across all datasets

Figure 2 presents a line graph of PofB@20% across all project pairs for the five models. IFA, or Initial False Alarms, measures how many clean (non-defective) modules are encountered before the first defective one in the ranking. Table IV lists IFA values for all experiments, comparing the proposed method against EALTR, EALR, CBS+, and EASC.

TABLE IV. THE IFA VALUE ON EACH CROSS-VERSION EXPERIMENT WHEN INSPECTING THE TOP 20%

Train	Test	Proposed Methodology	EALTR	EALR	CBS+	EASC
Camel v1.0	Camel v1.2	2	6	18	2	2
Camel v1.2	Camel v1.4	0	1	52	8	18
Camel v1.4	Camel v1.6	0	5	3	1	3
Ivy v1.1	Ivy v1.4	3	23	3	1	1
Ivy v1.4	Ivy v2.0	21	15	8	31	11
JEdit v3.2	JEdit v4.0	0	10	2	2	2
JEdit v4.0	JEdit v4.1	5	1	13	2	3
JEdit v4.1	JEdit v4.3	5	5	119	116	28
Log4j v1.0	Log4j v1.1	4	1	1	7	3
Log4j v1.1	Log4j v1.2	0	1	0	0	0
Lucene v2.0	Lucene v2.2	6	1	1	2	1
Lucene v2.2	Lucene v2.4	1	1	0	0	1
POI v1.5	POI v2.0	18	6	22	10	12
POI v2.0	POI v2.5	0	5	20	1	1
POI v2.5	POI v3.0	0	2	14	9	0
Synapsev1.0	Synapsev1.1	0	8	3	1	1
Synapse v1.1	Synapse v1.2	5	9	4	1	4
Velocity v1.4	Velocity v1.5	2	2	2	0	0
Velocity v1.5	Velocity v1.6	0	6	20	11	10
Xalan v2.4	Xalan v2.5	1	1	6	0	0
Xalan v2.5	Xalan v2.6	25	11	5	4	4
Xalan v2.6	Xalan v2.7	0	1	0	0	0
Xerces v1.2	Xerces v1.3	18	1	7	25	16
Xerces v1.3	Xerces v1.4	0	1	0	0	0
Xerces v1.4	Xerces v(init)	12	1	2	11	14

As shown in the results, the IFA results, presented in Table IV, highlight the effectiveness of the proposed methodology across multiple software projects. In several cases, the proposed method achieved the best performance compared to baseline approaches. In Camel v1.2- v1.4 and POI v2.0-v2.5, our method reached 0 and showed strong performance. This highlights the effectiveness of the ranking strategy. Similarly, in JEdit 3.2-v4.0 and Lucene v2.2-v2.4 bugs were discovered after one modules, which demonstrate efficient prioritization under limited effort. Although in some projects, such as Xalan 2.5- v2.6 and Xerces v1.2-v1.3, our method does not perform well, overall, the results confirm that the proposed approach generally achieves faster defect discovery while maintaining robustness across different software systems.

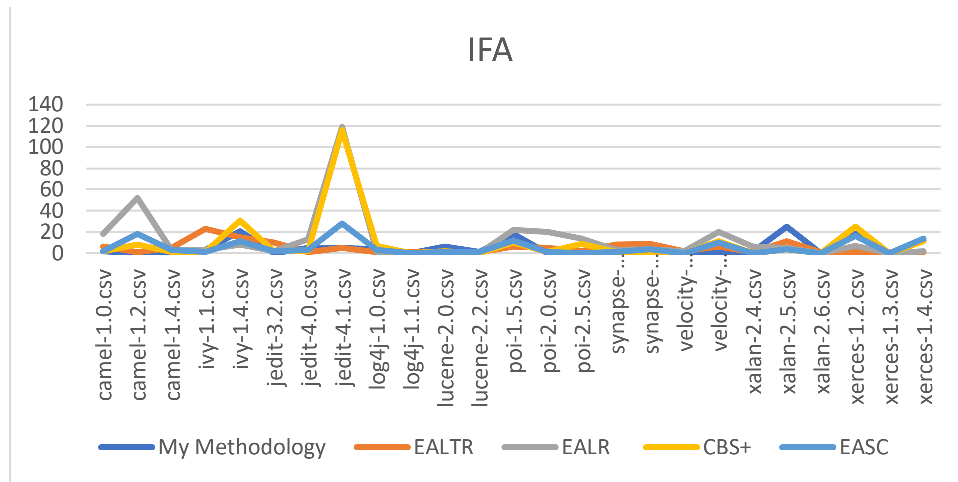

Figure 3. Comparison of IFA performance across all datasets

Figure 3 demonstrates a line graph comparing the IFA values of the four models across all project pairs. Popt is a widely used metric that measures how close the ranking is to the optimal ranking, which places the most defective and least costly modules first, and is calculated using the area under the cumulative lift curve. Table V presents the Popt values across various project version pairs.

TABLE V. THE POPT VALUE ON EACH CROSS-VERSION EXPERIMENT WHEN INSPECTING THE TOP 20%

Train	Test	Proposed Methodology	EALTR	EALR	EASC	CBS+
Camel v1.0	Camel v1.2	0.49	0.53	0.60	0.395	0.671185
Camel v1.2	Camel v1.4	0.74	0.63	0.69	0.7619	0.642352
Camel v1.4	Camel v1.6	0.72	0.62	0.63	0.6079	0.536616
Ivy v1.1	Ivy v1.4	0.53	0.58	0.64	0.5967	0.564543
Ivy v1.4	Ivy v2.0	0.49	0.51	0.54	0.4935	0.50919
JEdit v3.2	JEdit v4.0	0.85	0.73	0.79	0.7693	0.725104
JEdit v4.0	JEdit v4.1	0.90	0.76	0.75	0.7065	0.720095
JEdit v4.1	JEdit v4.3	0.84	0.66	0.78	0.5576	0.454766
Log4j v1.0	Log4j v1.1	0.78	0.78	0.66	0.6462	0.653127
Log4j v1.1	Log4j v1.2	0.49	0.55	0.40	0.348	0.179366
Lucene v2.0	Lucene v2.2	0.94	0.77	0.75	0.5689	0.51591
Lucene v2.2	Lucene v2.4	0.99	0.70	0.82	0.6264	0.707064
POI v1.5	POI v2.0	0.65	0.64	0.56	0.5173	0.472075
POI v2.0	POI v2.5	0.36	0.73	0.58	0.29	0.424253
POI v2.5	POI v3.0	0.86	0.74	0.85	0.7224	0.72958
Synapsev1.0	Synapsev1.1	0.55	0.61	0.47	0.5063	0.515199
Synapse v1.1	Synapse v1.2	0.62	0.56	0.52	0.5121	0.538925
Velocity v1.4	Velocity v1.5	1.00	0.89	0.95	0.9469	0.904512
Velocity v1.5	Velocity v1.6	1.00	0.80	0.87	0.7977	0.810771
Xalan v2.4	Xalan v2.5	0.48	0.83	0.68	0.5199	0.545925
Xalan v2.5	Xalan v2.6	0.95	0.78	0.81	0.6654	0.591841
Xalan v2.6	Xalan v2.7	1.00	0.93	0.93	0.4236	0.404817
Xerces v1.2	Xerces v1.3	0.82	0.66	0.79	0.5789	0.708768
Xerces v1.3	Xerces v1.4	1.00	0.72	0.73	0.4338	0.46178
Xerces v1.4	Xerces v(init)	0.83	0.87	0.93	0.4369	0.412584

As shown in Table V. In several cases, the method achieved substantial improvements compared to baseline approaches. For example, in *JEdit v4.0 → v4.1*, our method scored 0.90, which is higher than EALTR (0.76), EALR (0.75), CBS+ (0.70), and EASC (0.72). Similarly, in *Lucene v2.2 → v2.4*, our method achieved 0.99, clearly outperforming all the baselines. In several cases, the proposed approach achieved the maximum score of 1.0, including *Velocity v1.4 → v1.5*, *Velocity v1.5 → v1.6*, *Xalan v2.6 → v2.7*, and *Xerces v1.3 → v1.4*. These results demonstrate that the method can prioritize defective modules almost perfectly under effort constraints. Although there are

a few cases where baselines performed better, like *Xalan v2.4 → v2.5,* where EALTR (0.83) and EALR (0.68) surpassed our method (0.48), overall, the proposed methodology consistently matches or exceeds the alternatives. These outcomes confirm that optimizing the scoring function with PSO, combined with the re-ranking strategy, improves the ranking quality of software modules.

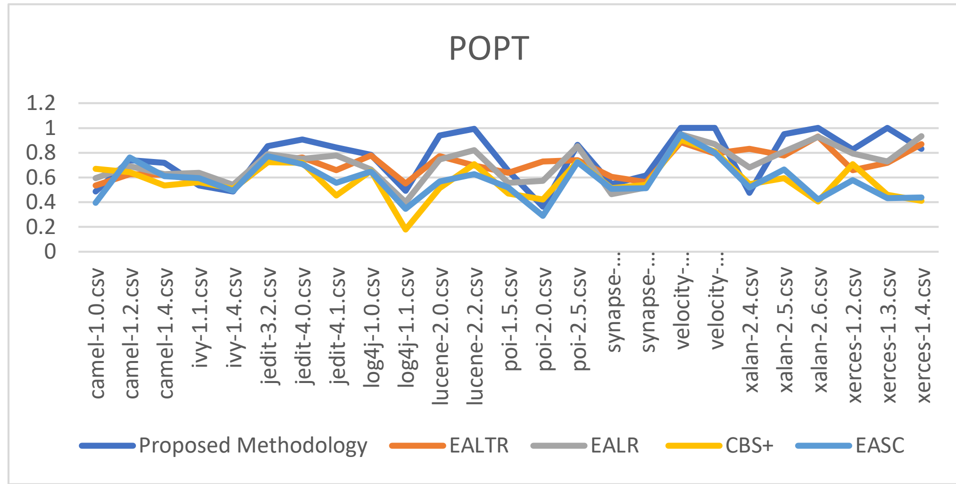

Figure 4. Comparison of POPT performance across all datasets

Figure 4 displays a line plot comparing the Popt values of the three models across all project pairs. The line representing our method remains consistently higher or competitive in nearly all scenarios.

TABLE VI. THE RECALL@20% VALUE ON EACH CROSS-VERSION EXPERIMENT WHEN INSPECTING THE TOP 20%

Train	Test	Proposed methodology	EALTR	EALR	CBS+	EASC
Camel v1.0	Camel v1.2	0.241	0.180077	0.26	0.421296	0.2222
Camel v1.2	Camel v1.4	0.441	0.343284	0.33	0.333333	0.5241
Camel v1.4	Camel v1.6	0.319	0.288	0.4	0.308511	0.25
Ivy v1.1	Ivy v1.4	0.312	0.388889	0.31	0.3125	0.375
Ivy v1.4	Ivy v2.0	0.175	0.303571	0.25	0.275	0.175
JEdit v3.2	JEdit v4.0	0.32	0.278761	0.08	0.466667	0.4667
JEdit v4.0	JEdit v4.1	0.367	0.40553	0.24	0.443038	0.3671
JEdit v4.1	JEdit v4.3	0.727	0.416667	0.36	0.090909	0.1818
Log4j v1.0	Log4j v1.1	0.27	0.453488	0.27	0.324324	0.3514
Log4j v1.1	Log4j v1.2	0.249	0.267068	0.43	0.21164	0.1534
Lucene v2.0	Lucene v2.2	0.403	0.318841	0.4	0.326389	0.3472
Lucene v2.2	Lucene v2.4	0.488	0.235759	0.5	0.492611	0.4335
POI v1.5	POI v2.0	0.378	0.358974	0.24	0.27027	0.2703
POI v2.0	POI v2.5	0.222	0.417339	0.17	0.185484	0.1089
POI v2.5	POI v3.0	0.459	0.368	0.44	0.476868	0.4698
Synapsev1.0	Synapsev1.1	0.217	0.232323	0.3	0.216667	0.2167
Synapse v1.1	Synapse v1.2	0.279	0.296552	0.28	0.267442	0.1977
Velocity v1.4	Velocity v1.5	0.676	0.465257	0.61	0.683099	0.6901
Velocity v1.5	Velocity v1.6	0.603	0.389474	0.41	0.551282	0.5641
Xalan v2.4	Xalan v2.5	0.217	0.527307	0.4	0.276486	0.1111
Xalan v2.5	Xalan v2.6	0.526	0.4688	0.43	0.40146	0.4331
Xalan v2.6	Xalan v2.7	0.631	0.44188	0.49	0.307606	0.3118
Xerces v1.2	Xerces v1.3	0.551	0.466321	0.45	0.449275	0.1594
Xerces v1.3	Xerces v1.4	0.35	0.442982	0.32	0.328638	0.0984
Xerces v1.4	Xerces v(init)	0.519	0.748503	0.6	0.342466	0.3766

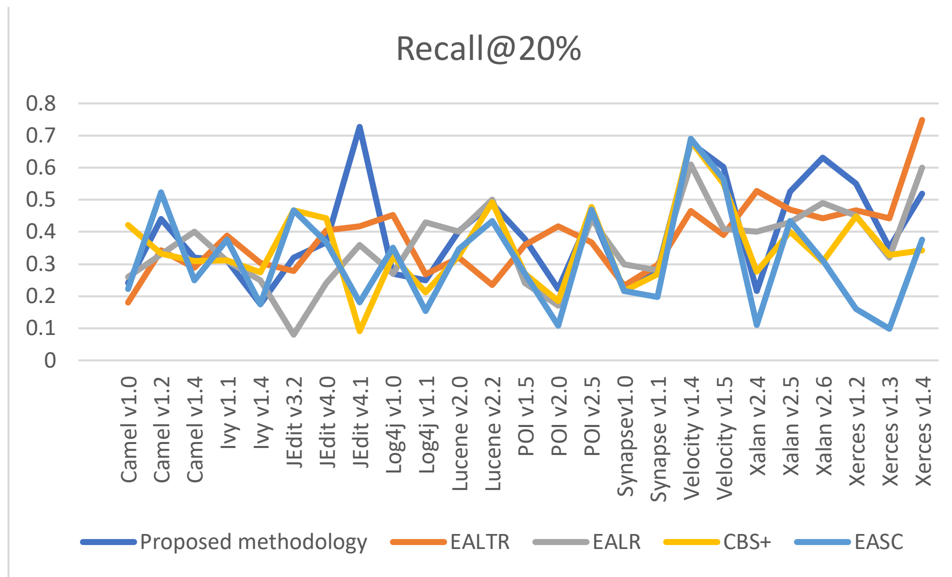

Figure 5. Comparison of Recall@20% performance across all datasets

Table VI presents the Recall@20% results for the proposed methodology compared with EALTR, EALR, CBS+, and EASC across multiple software projects. The proposed approach generally achieves higher Recall@20% values, showing its effectiveness in detecting a large portion of defects within the top 20% of inspection effort. For instance, in Camel v1.2–v1.4, it reaches 0.441, surpassing EALTR (0.3433) and EALR (0.33), while in Velocity v1.4–v1.5 and v1.5–v1.6, Recall@20% values of 0.676 and 0.603 demonstrate that most defects are captured by inspecting only a small fraction of the code. Some datasets, such as Xalan v2.4–v2.5 (0.217), show lower Recall@20%, indicating that defects are distributed in modules that require more inspection effort to detect. Overall, these results confirm that predicting total defects per module combined with effort-aware ranking allows testers to find more defects early and focus on modules that provide the highest defect yield per unit of effort.

TABLE VII. THE PRECISION@20% VALUE ON EACH CROSS-VERSION EXPERIMENT WHEN INSPECTING THE TOP 20%

Train	Test	Proposed Methodology	EALTR	EALR	CBS+	EASC
Camel v1.0	Camel v1.2	0.456	0.329897	0.39	0.354086	0.4706
Camel v1.2	Camel v1.4	0.151	0.114878	0.12	0.195122	0.3568
Camel v1.4	Camel v1.6	0.181	0.150735	0.22	0.242678	0.4608
Ivy v1.1	Ivy v1.4	0.043	0.051095	0.06	0.066667	0.0759
Ivy v1.4	Ivy v2.0	0.119	0.064516	0.06	0.106796	0.0588
JEdit v3.2	JEdit v4.0	0.522	0.165829	0.11	0.360825	0.4667
JEdit v4.0	JEdit v4.1	0.333	0.190217	0.17	0.472973	0.58
JEdit v4.1	JEdit v4.3	0.046	0.02963	0.03	0.008264	0.0274
Log4j v1.0	Log4j v1.1	0.476	0.424242	0.56	0.5	0.7647
Log4j v1.1	Log4j v1.2	0.959	0.979167	0.94	0.930233	0.9062
Lucene v2.0	Lucene v2.2	0.592	0.702703	0.52	0.671429	0.7353
Lucene v2.2	Lucene v2.4	0.556	0.54	0.53	0.649351	0.5714
POI v1.5	POI v2.0	0.087	0.076087	0.05	0.071942	0.0769
POI v2.0	POI v2.5	0.414	0.6	0.43	0.528736	0.75
POI v2.5	POI v3.0	0.592	0.55642	0.58	0.647343	0.6911
Synapsev1.0	Synapsev1.1	0.481	0.233766	0.28	0.236364	0.3095
Synapse v1.1	Synapse v1.2	0.293	0.26506	0.26	0.338235	0.5152
Velocity v1.4	Velocity v1.5	0.611	0.612403	0.6	0.617834	0.6242
Velocity v1.5	Velocity v1.6	0.388	0.291971	0.24	0.338583	0.386
Xalan v2.4	Xalan v2.5	0.568	0.435028	0.43	0.614943	0.6515
Xalan v2.5	Xalan v2.6	0.355	0.363184	0.34	0.517241	0.5298
Xalan v2.6	Xalan v2.7	0.983	0.983645	0.98	0.996377	1
Xerces v1.2	Xerces v1.3	0.133	0.117333	0.13	0.192547	0.1803
Xerces v1.3	Xerces v1.4	0.769	0.708609	0.72	0.886076	1
Xerces v1.4	Xerces v(init)	0.374	0.470149	0.42	0.260417	0.2959

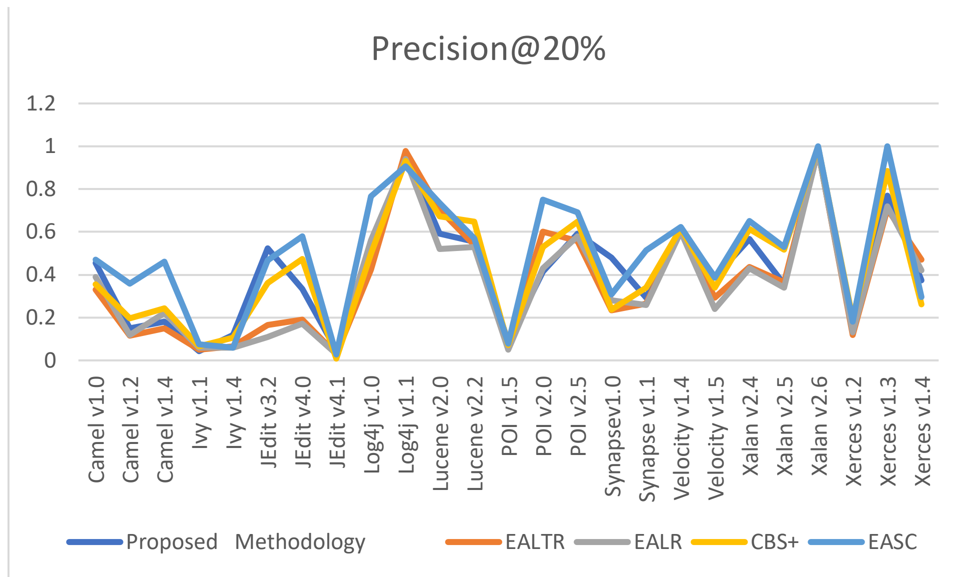

Figure 6. Comparison of Precision@20% performance across all datasets

Table VII reports the Precision@20% results of the proposed methodology and compares them with EALTR, EALR, CBS+, and EASC across multiple software projects. The proposed method consistently achieves higher Precision@20% values, indicating its ability to prioritize modules that yield the most defects for the least inspection effort. For example, in Camel v1.0–v1.2, the proposed approach reaches 0.456, outperforming EALTR (0.3299) and EALR (0.39), while in some datasets, such as Xalan v2.6–v2.7 and Xerces v1.3–v1.4, Precision@20% reaches 1.0, meaning all defects within the top 20% of effort were captured. This occurs because the 20% LOC budget is exhausted after inspecting a few high-priority modules, which is a known characteristic of LOC-based effort-aware evaluation. In other datasets, Precision@20% is lower, indicating that the top-ranked modules contain fewer defects relative to the inspection effort. Overall, these results demonstrate that predicting total defects per module combined with effort-aware ranking allows testers to find the maximum number of defects while minimizing the inspection effort.Precision@20% saturates at 1.0 on Xalan-2.7 due to the extreme class imbalance, where 98.8% of modules are defective. This behavior reflects dataset characteristics rather than model bias.

TABLE VIII. AVERAGE VALUE OF POFB@20%, IFA, AND POPT METRICS ACROSS ALL SOFTWARE DATASETS

Average	Proposed Methodology	EALTR	EALR	CBS+	EASC
PofB@20%	0.402	0.371	0.32	0.35	0.31
IFA	5.1	4.96	9.08	9.80	5.40
Popt	0.756	0.701	0.28	0.57	0.57
Recall@20%	0.398	0.38	0.36	0.35	0.314
Precision@20%	0.419	0.37	0.37	0.43	0.499

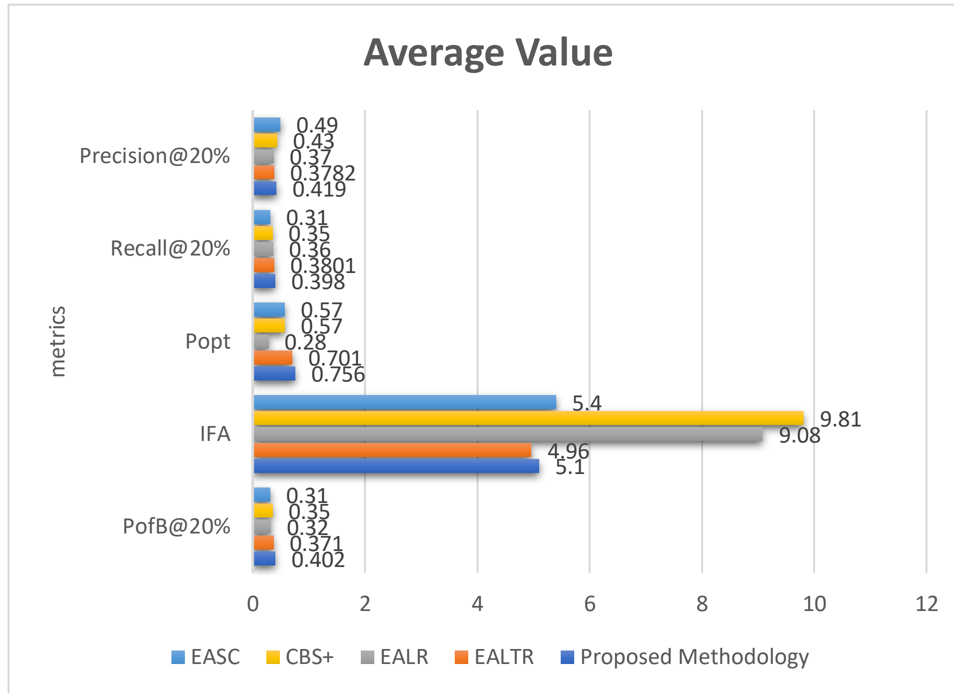

Figure 7. Average values of POFB@20%, IFA, and POPT metric across all software datasets

. The average results across all projects are shown in Table VI. Overall, the averages prove that the proposed approach balances early defect detection and efficient effort usage better than the compared method. Figure 5 shows us the bar diagram which compares the average performance of the proposed methodology with four existing techniques (EALTR, EALR, CBS+, and EASC) across three effort-aware evaluation metrics.

VI. Conclusion

This study introduced an effort-aware defect prediction framework designed to help testers detect more bugs with less inspection effort. The method combined a Random Forest regressor with a PSO-optimized scoring function that balances predicted bug counts against code size, while a re-ranking step improved early defect discovery. Evaluated on 25 cross-version datasets from open-source Java projects, the approach achieved strong results with an average PofB@20% of 0.402, Popt of 0.756, and IFA of 5.1, outperforming baselines such as EALTR, EALR, CBS+, and EASC. These findings suggest that the Random Forest Regressor, when combined with effort-aware optimization, can significantly improve the ranking of modules. Since the current scoring function uses a Random Forest, in the future, exploring more flexible models like adaptive or non-linear functions that better reflect project-specific patterns or real-time feedback from testers.

References

[1] X. Yu et al., “Finding the best learning to rank algorithms for effort-aware defect prediction,” Information and Software Technology, vol. 157, p. 107165, May 2023, doi: 10.1016/j.infsof.2023.107165.

[2] M. I. Hossain, “Software Development Life Cycle (SDLC) Methodologies for Information Systems Project Management,” vol. 5, no. 5, 2023.

[3] “Software Quality Assurance - Software Engineering - GeeksforGeeks.” Accessed: Jan. 28, 2025. [Online]. Available: https://www.geeksforgeeks.org/software-engineering-software-quality-assurance/

[4] A. Bertolino, “Software Testing Research: Achievements, Challenges, Dreams,” in Future of Software Engineering (FOSE '07), Minneapolis, MN, USA: IEEE, May 2007, pp. 85–103. doi: 10.1109/FOSE.2007.25.

[5] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, “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, Amsterdam The Netherlands: ACM, Aug. 2009, pp. 91–100. doi: 10.1145/1595696.1595713.

[6] T. Mende and R. Koschke, “Effort-Aware Defect Prediction Models,” in 2010 14th European Conference on Software Maintenance and Reengineering, Madrid: IEEE, Mar. 2010, pp. 107–116. doi: 10.1109/CSMR.2010.18.

[7] Y. Kamei et al., “A large-scale empirical study of just-in-time quality assurance,” IEEE Trans. Software Eng., vol. 39, no. 6, pp. 757–773, Jun. 2013, doi: 10.1109/TSE.2012.70.

[8] X. Yu et al., “Improving effort-aware defect prediction by directly learning to rank software modules,” Information and Software Technology, vol. 165, p. 107250, Jan. 2024, doi: 10.1016/j.infsof.2023.107250.

[9] Q. Huang, X. Xia, and D. Lo, “Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction,” Empir Software Eng, vol. 24, no. 5, pp. 2823–2862, Oct. 2019, doi: 10.1007/s10664-018-9661-2.

[10] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu, “Revisiting Supervised and Unsupervised Methods for Effort-Aware Cross-Project Defect Prediction,” IEEE Trans. Software Eng., vol. 48, no. 3, pp. 786–802, Mar. 2022, doi: 10.1109/TSE.2020.3001739.

[11] P. Yang et al., “Bug numbers matter: An empirical study of effort-aware defect prediction using class labels versus bug numbers,” Software: Practice and Experience, vol. 55, no. 1, pp. 49–78, 2025, doi: 10.1002/spe.. 3363.

[12] S. S. Reddy and S. Pabboju, “Enhancement of Defect Prediction Using Regression Learning Method for Quality Software Development,” International Journal of Intelligent Engineering & Systems, vol. 17, no. 6, p. 424, Nov. 2024, doi: 10.22266/ijies2024.1231.33.

[13] J. Çarka, M. Esposito, and D. Falessi, “On effort-aware metrics for defect prediction,” Empir Software Eng, vol. 27, no. 6, p. 152, Nov. 2022, doi: 10.1007/s10664-022-10186-7.

[14] Y. Kamei et al., “A large-scale empirical study of just-in-time quality assurance,” IEEE Trans. Software Eng., vol. 39, no. 6, pp. 757–773, Jun. 2013, doi: 10.1109/TSE.2012.70.

[15] X. Yu, L. Liu, L. Zhu, J. W. Keung, Z. Wang, and F. Li, “A multi-objective effort-aware defect prediction approach based on NSGA-II,” Applied Soft Computing, vol. 149, p. 110941, 2023.

[16] X. Du et al., “CoreBug: Improving Effort-Aware Bug Prediction in Software Systems Using Generalized k-Core Decomposition in Class Dependency Networks,” Axioms, vol. 11, no. 5, p. 205, Apr. 2022, doi: 10.3390/axioms11050205.

[17] R. Malhotra, R. Kapoor, P. Saxena, and P. Sharma, “SAGA: A Hybrid Technique to handle Imbalanced Data in Software Defect Prediction,” in 2021 IEEE 11th IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE), Apr. 2021, pp. 331–336. doi: 10.1109/ISCAIE51753.2021.9431842.

[18] W. Li, W. Zhang, X. Jia, and Z. Huang, “Effort-aware semi-supervised just-in-time defect prediction,” Information and Software Technology, vol. 126, p. 106364, 2020.

[19] X. Yang, H. Yu, G. Fan, and K. Yang, “DEJIT: A Differential Evolution Algorithm for Effort-Aware Just-in-Time Software Defect Prediction,” Int. J. Soft. Eng. Knowl. Eng., vol. 31, no. 03, pp. 289–310, Mar. 2021, doi: 10.1142/s0218194021500108.

[20] X. Yu et al., “Finding the best learning to rank algorithms for effort-aware defect prediction,” Information and Software Technology, vol. 157, p. 107165, May 2023, doi: 10.1016/j.infsof.2023.107165.

[21] F. Li, W. Lu, J. W. Keung, X. Yu, L. Gong, and J. Li, “The impact of feature selection techniques on effort-aware defect prediction: An empirical study,” IET Software, vol. 17, no. 2, pp. 168–193, Apr. 2023, doi: 10.1049/sfw2.12099.

[22] F. Li, P. Yang, J. W. Keung, W. Hu, H. Luo, and X. Yu, “Revisiting ‘revisiting supervised methods for effort-aware cross-project defect prediction,’” IET Software, vol. 17, no. 4, pp. 472–495, Aug. 2023, doi: 10.1049/sfw2.12133.

[23] Y. Guo, M. Shepperd, and N. Li, “Improving classifier-based effort-aware software defect prediction by reducing ranking errors,” May 13, 2024, arXiv: arXiv:2405.07604. doi: 10.48550/arXiv.2405.07604.

[24] A. Boloori, A. Zamanifar, and A. Farhadi, “Enhancing software defect prediction models using metaheuristics with a learning to rank approach,” Discov Data, vol. 2, no. 1, p. 11, Nov. 2024, doi: 10.1007/s44248-024-00016-0.

[25] Y. Zhou et al., “How Far We Have Progressed in the Journey? An Examination of Cross-Project Defect Prediction,” ACM Transactions on Software Engineering and Methodology, vol. 27, no. 1, pp. 1–51, Jan. 2018, doi: https://doi.org/10.1145/3183339.

[26] S. Fatima, Z. Fatima, M.A. Hayat, M.H. Shahab, M.K. Meraj, R.M. Ibrahim, and S.M Muneeb, “Impact of Software Metrics on Software Quality using McCall Quality Model: In-Depth Analysis”, 2022.